Translating Database Queries to English for Enhancing Database Education

A Thesis
Presented to
The Academic Faculty

by

William J. Holton

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science

Georgia Institute of Technology
December 2015

Translating Database Queries to English for Enhancing Database Education

Approved by:

Dr. Mayur Naik, Advisor
College of Computing
Georgia Institute of Technology

Dr. Shamkant Navathe
College of Computing
Georgia Institute of Technology

Dr. Edward Omiecinski
College of Computing
Georgia Institute of Technology

Date Approved: December 4th, 2015

ACKNOWLEDGEMENTS

The completion of my M.S. in Computer Science would not be possible without my loving family, who have supported me throughout and always encouraged me to do more. For his constant aid and guidance throughout my education here at Georgia Tech, I would also like to express my utmost appreciation to Professor Mayur Naik, without whom my accomplishments would not have been possible. I could not imagine a more generous or patient advisor, and have been truly privileged to work alongside him. Further, I am indebted to Dr. Sumit Gulwani for being a fountain of knowledge from which this work drew heavily for both the initial direction as well as for continual suggestions towards its end. Finally, I would like to thank my research partner Jake Cobb for his help in building the tools and gathering the data necessary for the writing of this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

With the continuing integration of and reliance on technology by society, the demand for strong computer science curriculum has continued to grow. Databases permeate all sectors of industry and thus database programming, in particular, has high impact due to the volume and variety of students whose future careers depend on the subject; the undergraduate database course at Georgia Tech already hosts classes with over two-hundred students, with upwards of 10 different majors represented among them. With the advent of Massively Open Online Courses (MOOCs), enrollment is expected to increase sharply and be comprised of a much more diverse student body in terms of demographics, major, and educational and professional background. This situation poses two major challenges for database education going forward: 1) improving pedagogy to cater to a large and diverse student population and 2) scaling education up to handle much higher student-to-teacher ratios. Creating course materials and evaluating student performance are both tedious tasks for instructors that require creativity, hard work, and pedagogical experience. Therefore, these challenges require automated assistance and cannot be met by manual effort alone.

The thesis of this research is that database queries can be translated to corresponding English descriptions for the use in applications in intelligent tutoring (in particular, problem generation and feedback generation) for the subject domain of database query construction. To demonstrate this thesis, a rule-based graph-rewriting algorithm and a concrete set of rules for systematically transforming queries in a subset of SQL to English descriptions are presented. Further, through an implementation of this technique, this study demonstrates an evaluation of its performance on SQL queries from database textbooks.

CHAPTER I: INTRODUCTION

Databases permeate all sectors of industry. With the sheer volume of data being generated every day, the ability to manage and process it has become a highly sought skill [31]. The demand for this ability has led large numbers of individuals from diverse backgrounds to seek education in database technology. This influx has exacerbated the problem of providing personalized and interactive database education, in classrooms and online education platforms alike.

Computer-aided intelligent tutoring [14] aims to alleviate this problem by automating repetitive pedagogical tasks such as *problem generation* and *feedback generation*. This thesis seeks to help develop such tutoring for the subject domain of database query construction, an important topic in database education. In particular, it addresses a problem central to this task: how to automatically translate database queries into natural language descriptions, which from this point on will be interchangably refered to as *paraphrasing* database queries. Specifically, the work to be presented is a technique and system that translates SQL queries into English descriptions.

In contrast to the well-studied problem of synthesizing database queries from natural language descriptions [4, 21, 20, 18, 25, 26], paraphrasing database queries requires formalizing succinct and refined English descriptions of the kind seen in textbooks. Paraphrasing is challenging because it must simultaneously address the problems of extracting domain-specific knowledge and obeying the rules of the output language. While database queries are low-level implementation-specific expressions, human-written natural language descriptions are expressed in terms of a conceptual model of some real-world domain; this domain-specific conceptual knowledge must be extracted to generate language that is not a mechanical recitation of a query's syntax. At the same time, English has many complex rules for correctness that must be obeyed, e.g. subject-verb agreement, as well as opportu-

nities for reducing verbosity, e.g. by compounding subjects to avoid repetitive clauses. This work will illustrate these challenges using motivating examples in Chapter II and discuss applications in Chapter III.

The paraphrasing approach is a rule-based graph-rewriting technique, described in Chapter IV, which takes a conceptual model, an implementation model, a mapping between the conceptual and implementation models, and a database query as input. The inputs and output are all represented by a single graph structure. The technique systematically transforms our input query into an English output by repeated rule application, where each rule is defined by a matching predicate and a sequence of rewriting actions. These rules are examined in Chapter V. The design of this technique is motivated by several concerns. First, by taking a conceptual model as input, the rules may be written generally and thus are *highly reusable* across different model instances. Second, the system is *easily extensible* as new query constructs or translation strategies can be supported by defining new rules. Finally, the system addresses both domain knowledge and language-specific concerns *uniformly* by operating on a single graph structure, allowing rules to address either concern or both together.

This technique has been implemented into a working system for a subset of SQL. The system, described in Chapter VI, specifies the rules declaratively in Datalog, and evaluates them using the IRIS system [6]. In Chapter VII, the system is evaluated by using it to paraphrase SQL queries found in textbooks and variations thereof that simulate the database tutoring applications. Additionally, the aggregate complexity of rule evaluation for our experimental queries is analyzed in order to demonstrate its flexibility and generality.

1.1 Thesis Statement

The thesis of this research is that database queries can be translated to corresponding English descriptions for the use in applications in intelligent tutoring (in particular, problem generation and feedback generation) for the subject domain of database query construction.

1.2 Contributions

This work makes the following contributions:

1. A formal description of the problem of translating a database query into natural language.

2. A novel algorithm for translating a database query into a succinct natural language description.

3. Applications of this technology, specifically problem generation and feedback generation in the context of intelligent tutoring.

4. Experimental results that evaluate the ability of the system to produce succinct natural descriptions in the context of educational applications.

CHAPTER II: MOTIVATING EXAMPLE

To illustrate the challenges handled by the paraphrasing technique, this chapter will begin with a few example SQL queries and their translations. The system is rule-based, but this chapter focuses on the high-level concepts and actions of the system and leaves the low-level details of individual rules for later. These examples target a simple database from a textbook [12] that models a company. An annotated ER diagram is used for the conceptual model, a relational schema is used for the implementation model, and additionally, there is an ER-relational mapping to relate the two. Figure 1 shows the ER diagram for the company database. For brevity, only a subset of the relational schema that is relevant to the examples has been included:

```
employee(ssn, fname, minit, lname, address, super_ssn, dno)
department(dname, dnumber)
```

The ER-relational mapping maps attributes of entities in the ER diagram to columns of relations in the relational schema. For example, it maps the `Fname` attribute of the `Employee` entity to `employee.fname`. Additionally, it maps the relationship edges in the ER diagram to joins on keys in the relational schema. For example, there is a 1-to-N relationship between supervisors and the employees they supervise. The ER-relational mapping indicates that the join of two instances of the `employee` relation, `e1` and `e2`, on the condition `e1.ssn = e2.super_ssn`, maps to the `SUPERVISION` relationship, and that `e1` is in the `Supervisor` role, while `e2` is in the `Supervisee` role. The details of these inputs to the system are further discussed in Chapter IV, Section 1.1.

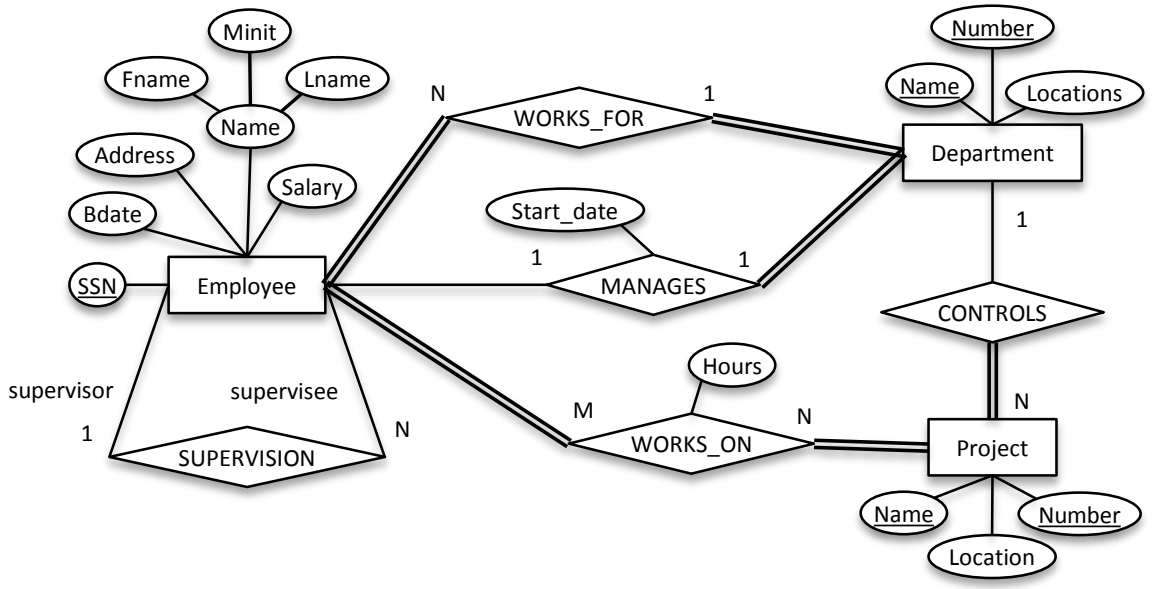

Figure 1: ER diagram for the company database.

For a very simple query such as

(1) `SELECT address FROM employee`

one could obtain the translation

(1) Select the address of each employee.

via a syntax-based approach. But this is not the case in general. The two main goals are *extracting domain-specific knowledge* and *producing high-quality natural language*, meaning the natural language is not overly verbose, is correct according to proscriptive grammar rules, and does not feel mechanical. However, these two concerns are not cleanly separated, as the following examples demonstrate.

Consider the query:

(2)
```
SELECT e.lname, s.lname
FROM employee e, employee s
WHERE e.super_ssn=s.ssn
```

If one was to take a purely syntax-based approach, they might end up with a translation similar to:

> Select the lname of each employee *e* and the lname of each employee *s* where *e*'s super ssn equals *s*' ssn.

Instead, the system uses the following domain-specific information to yield a better translation: *i)* the join condition `e.super_ssn = s.ssn` maps to a relationship in the ER diagram, where `s` is in the "supervisor" role, and *ii)* `employee.lname` maps to the `Employee.Lname` attribute, which is annotated with the label "last name". Using this domain-specific information, the system can produce the translation:

> Select the last name of each employee and the last name of the employee's supervisor.

While this translation is correct, the system is able to produce an even more concise and natural translation by using the following information: *i)* the possessive phrase "employee's supervisor" can use a possessive pronoun instead, *ii)* the `Employee` entity is annotated as a person, so the possessive pronoun should be "their", and *iii)* the two prepositional phrases start with the same expression ("last name") and may be merged. This yields the final translation

(2) Select the last name of each employee and their supervisor.

Note that the use of pronouns and possessives in general involves anaphoric referencing [11] and must be used carefully to avoid ambiguity. In fact, some natural-language-specific logic was already needed to decide to use "employee's supervisor" instead of stating that "`s` supervises `e`."

As a final example, consider the query

(3)
```
SELECT e.fname, e.minit, e.lname, e.address
FROM employee e, department d
WHERE d.dname='Research' AND d.dnumber=e.dno
```

and the translation

(3) Select the name and address of each employee who works for the Research department.

Several different pieces of information are discovered by the system and used to produce this translation. First, the three columns `fname`, `minit`, and `lname` are collapsed into the single composite attribute `Name` from the ER diagram. Second, the `Department.Dname` attribute is a "describing attribute", as annotated in the ER diagram, and thus the literal value "Research" may be used to describe the entity. The resulting phrase, "Research department" is known as a nominal compound [11]. Third, because the `Department.Name` attribute is a key, there is only one such department, hence the choice of "the" instead of "a" as the determiner. Fourth, the key comparison `d.dnumber = e.dno` maps to the `WORKS_FOR` relationship and so the relationship "works for" is expressed instead of the key equality. Finally, the `Employee` entity is annotated as a person, so the term "who" is used instead of "that" when expressing the `WORKS_FOR` relationship.

CHAPTER III: APPLICATIONS

There are a number of applications for natural language descriptions of SQL queries, particularly in the context of education. These applications include *i)* feedback generation, for automatic aid with incorrect queries, *ii)* problem generation, for fully automatic creation of new problems, *iii)* guided problem presentation, to reinforce learning concepts, *iv)* debugging, for both novice and regular users, and *v)* as a complement to natural language interfaces to relational database systems.

3.1 Feedback Generation

In the education domain, query paraphrasing is useful for *feedback generation*, so that a student providing an incorrect query can be given a description of the answer they gave to compare with the original problem description. For example, the original query may be

(4)
```
SELECT e.fname, s.salary
FROM employee e, employee s
WHERE e.super_ssn=e.ssn
```

with the translation

(4) Select the first name of each employee and the salary of their supervisor.

If the student gives the answer as

(5)
```
SELECT e.fname, s.salary
FROM employee e, employee s
WHERE e.ssn=s.super_ssn
```

then it would be translated as

(5) Select the salary of each employee and the first name of their supervisor.

which indicates the mistake of reversing the relationships between `e` and `s`. Feedback has been shown to enhance learner performance in prior work, e.g. [8].

3.2 Problem Generation

Query paraphrasing is also useful for *problem generation*. An instructor may quickly generate new problems by simply writing the queries and allowing paraphrasing to create the problem descriptions. Furthermore, a complementary technique may be used to generate the SQL queries themselves, so that problem generation is fully automatic outside of the ER diagram and schema designs. Consider a scenario where the instructor has determined a set of query templates that represent specific concepts they want students to learn. A problem generation technique can create queries for specific data models based on these templates without requiring a data-model-specific description of each one from the instructor. For example, to teach simple selection and filtering, the template could be

```
SELECT attribute1 FROM entity
WHERE attribute2 op constant
```

with concrete instances such as

(6)
```
SELECT pname FROM project
WHERE plocation = 'Texas'
```

(7)
```
SELECT lname FROM employee
WHERE salary > 50000
```

Query paraphrasing produces the descriptions

(6) Select the name of each project located in Texas.

(7) Select the last name of each employee whose salary is greater than $50,000.

3.3 Guided Problem Presentation

In *guided problem presentation*, the instructor wants to enforce specific concepts but needs a variety of problems to prevent rote memorization or copying. Paraphrasing can produce the problem descriptions for variations on a template query structure that may be systematically produced by another technique. If a student fails to answer a query correctly, the structure may be simplified and paraphrasing will produce the new problem description. To illustrate this scenario, consider the query

(8)
```
SELECT e.salary, d.dname
FROM employee e, department d
WHERE d.dnumber = e.dno
AND e.fname = 'Ahmad'
```

with the translation

(8) Select the salary of each employee whose first name is "Ahmad" and the name of the department they work for.

This query can be simplified by removing the join condition, yielding the following query and translation:

(9)
```
SELECT salary FROM employee e
WHERE fname = 'Ahmad'
```

(9) Select the salary of each employee whose first name is "Ahmad."

In this case, the join is eliminated while the filtering condition on the `employee` table remains. In Chapter VII, we present several experiments that demonstrate the value of paraphrasing for this application.

3.4 Query Paraphrasing

Query paraphrasing is useful for both novice and regular query authors for verifying whether a query is having the desired effect. Incorrect queries are often the result of small syntactic differences that have large semantic implications [7, 22]. In [13], seven real-world example errors are given; only one requires changing two expressions and the rest require only a single expression to be added or modified. Consider this example from Welty [32], where the query

(10)
```
SELECT HEAD FROM DEPARTMENT
    WHERE DEPT = BUILDING
```

translates to

(10) Select the head of each department where its name is equal to its building.

whereas the query

(11)
```
SELECT HEAD FROM DEPARTMENT
    WHERE DEPT = 'BUILDING'
```

translates to

(11) Select the head of the "BUILDING" department.

3.5 Natural Language Database Interfaces

Finally, query paraphrasing complements systems that provide a natural language interface to backend database systems. These systems accept natural language requests and translate them into queries in a formal language, such as SQL, for the backend database. Due to ambiguity, anaphoric references, limited linguistic concept support, etc, it is not always clear that the generated query accurately reflects the user's intent [15]. Query paraphrasing can translate the generated query back into natural language, which allows the user to gauge whether or not the SQL query appears to accurately reflect their intended meaning.

CHAPTER IV: ALGORITHM

The algorithm takes three inputs: *i)* a conceptual model (annotated ER diagram), *ii)* a mapping from the conceptual model to an implementation model (ER-relational mapping), and *iii)* a logical form (SQL query[1]) to translate to natural language. Our algorithm is rule-based and operates on a property graph $G = G_{ER} \oplus G_Q$ that is composed of a property graph G_{ER}, encoding the ER diagram and ER-relational mapping, and a property graph G_Q, encoding the query and verbalization, which we describe in Chapter IV, Section 1.

Formally, a *property graph* $G = (V, E, \mu)$ is a directed, attributed multigraph[2] that consists of a set of nodes V, a multiset of directed edges E, and a property mapping function $\mu : (V \cup E) \times \Sigma_K \to \Sigma_P$, where Σ_K is a set of property names and Σ_P is a set of property values. We refer to the value of the `node-type` or `edge-type` properties as the *type* of a node or edge, respectively, and denote by $\Sigma_L \subseteq \Sigma_P$ the set of type names. We write $n_1 \xrightarrow{\ell} n_2$ to denote an edge from node n_1 to n_2 of type ℓ.

4.1 Inputs

4.1.1 ER Diagram

We require a conceptual model to reason about the domain of discourse. For this purpose, we use the ER diagram from the Entity-Relational Model [9] as it is a standard data model for relational database systems. An ER diagram is typically defined as an undirected multigraph with labeled, typed and attributed nodes, and typed and attributed edges. We define it as a property graph $G_{ER} = (V_{ER}, E_{ER}, \mu_{ER})$ as follows. V_{ER} and E_{ER} are the nodes and edges

[1] We focus on education and support a limited subset of SQL that does not include aggregation or nested queries.

[2] We omit the edge-labeling of property graphs from [28].

of the ER diagram, respectively. All nodes have a `name` property with the object's name. The type of each node in V_{ER} is one of `entity`, `relationship`, or `attribute`. For simplicity, we refer to nodes $v \in V_{ER}$ of type `entity` as entities or `entity` nodes, and similarly for the other types. To avoid confusion, we refer to `attribute` nodes as ER-attributes. Each edge in E_{ER} has a type corresponding to the node types it connects, which is one of:

- `entity-relationship`,
- `entity-attribute`,
- `relationship-attribute`, or
- `attribute-attribute`

ER diagram edges are normally undirected; we direct the edges based on the connected node types in an arbitrary but consistent direction, e.g. from relationships to entities. ER-attributes have the following standard Boolean properties:

- `composite`: If the ER-attribute is composed of other ER-attributes.
- `key`: If the ER-attribute is a key, meaning a unique identifier of the parent.

Edges of type `entity-relationship` have properties:

- `cardinality`: a positive integer or the sentinel value `N`.
- `role-label`: an optional English expression for the entity attached to this edge.

There are exactly two `entity-relationship` edges for each `relationship` node. `attribute-attribute` edges require that at least one of the `attribute` nodes is composite.

4.1.2 ER Diagram Extensions

We extend the definition of the ER diagram to allow some rules to make more intelligent translations than would otherwise be possible in the following way. `entity` nodes have a

property `entity-type` indicating whether the entity is a `person`, `place` or `thing`. Entities and ER-attributes have `singular` and `plural` properties, and relationships have a `verb` property for the basic English expressions of the modeled objects. ER-attributes also have a property `data-type`, indicating the data type of the attribute, e.g. currency, and optionally `describing`, indicating fixed values of the attribute can be used to reference the parent entity.

4.1.3 ER-relational Mapping

The ER-relational mapping connects the ER diagram model to the particular relational schema used. The relational schema is defined by a set of tables, where each table consists of a set of columns. Let Σ_T be the set of table names and Σ_C be the set of column names, then $S \subseteq \Sigma_T \times \Sigma_C$ defines the existing tables and columns in the schema. Let $K \subseteq S$ be the primary and foreign keys of the schema S, where K_t denotes the keys of table t. Without loss of generality, each key $k \in K$ is a single table-column pair $k = (t, c)$. The ER-relational mapping $M = (M_{ATTR}, M_{FKJOIN}, M_{STJOIN})$ is defined by the following partial functions:

- $M_{ATTR} : V_{ER} \nrightarrow S$, maps non-composite ER-attributes to relational columns.
- $M_{FKJOIN} : V_{ER} \nrightarrow K^2$, maps relationships to a foreign-key join between a key-pair $(k_1, k_2) \in K^2$, where k_1 is a primary key and k_2 is a foreign key.
- $M_{STJOIN} : V_{ER} \nrightarrow K^4$, maps a relationship to a separate-table join that is defined using two pairs of keys $(k_1, k_2, k_3, k_4) \in K^4$, where $k_1 \in K_{t_1}, k_4 \in K_{t_2}$ are each primary keys and $k_2, k_3 \in K_{t_3}$, $k_2 \neq k_3$ are corresponding foreign keys in an intermediate table.

One of M_{FKJOIN} or M_{STJOIN} is defined for every relationship, but not both. Note that the foreign-key mapping style cannot be used for many-to-many relationships. The ER-relational mapping is included in G as shown in Chapter IV, Section 3.

4.1.4 SQL Query

The SQL query is parsed as an AST using an off-the-shelf parser and encoded as discussed in Chapter IV, Section 2. We adhere to the following subset of SQL:

⟨*QUERY*⟩ ::= ⟨*SELECT*⟩ ⟨*FROM*⟩ ⟨*WHERE*⟩

⟨*SELECT*⟩ ::= ⟨*SELECT-ALL*⟩ | ⟨*SELECT-DISTINCT*⟩

⟨*SELECT-ALL*⟩ ::= 'SELECT' ⟨*selectList*⟩

⟨*SELECT-DISTINCT*⟩ ::= 'SELECT' 'DISTINCT' ⟨*selectList*⟩

⟨*selectList*⟩ ::= ⟨*selectTerm*⟩

⟨*selectTerm*⟩ ::= ⟨*tablecolumn*⟩ | ⟨*tablecolumn*⟩ ⟨*selectTerm*⟩

⟨*FROM*⟩ ::= 'FROM' ⟨*fromList*⟩

⟨*fromList*⟩ ::= ⟨*tableTerm*⟩

⟨*tableTerm*⟩ ::= ⟨*table*⟩ | ⟨*table*⟩ 'JOIN' ⟨*tableTerm*⟩

⟨*table*⟩ ::= ⟨*tablename*⟩ | ⟨*tablename*⟩ 'AS' ⟨*tablealias*⟩

⟨*tablecolumn*⟩ ::= ⟨*tablealias*⟩ '.' ⟨*columnname*⟩

⟨*WHERE*⟩ ::= 'WHERE' ⟨*whereList*⟩

⟨*whereList*⟩ ::= ⟨*whereTerm*⟩

⟨*whereTerm*⟩ ::= ⟨*binaryCompare*⟩ | ⟨*and*⟩ | ⟨*or*⟩

⟨*and*⟩ ::= ⟨*whereTerm*⟩ 'AND' ⟨*whereTerm*⟩

⟨*or*⟩ ::= ⟨*whereTerm*⟩ 'OR' ⟨*whereTerm*⟩

⟨*binaryCompare*⟩ ::= ⟨*tablecolumn*⟩ ⟨*operator*⟩ ⟨*tablecolumn*⟩ | ⟨*tablecolumn*⟩ ⟨*operator*⟩ ⟨*value*⟩

⟨*operator*⟩ ::= '<' | '<=' | '>' | '>=' | '=' | '<>'

⟨*value*⟩ ::= ⟨*bool*⟩ | ⟨*number*⟩ | ⟨*string*⟩

⟨*value*⟩ is a boolean, numeric or string constant. ⟨*tablename*⟩, ⟨*tablealias*⟩ and ⟨*columnname*⟩ are identifiers.

Figure 2: The SQL grammar.

4.2 Symbolic State

The symbolic state encodes the query and verbalization as the property graph $G_Q = (V_Q, E_Q, \mu_Q)$. Nodes in V_Q are *symbolic tokens* of type `symbolic`. There are two classes of edges in E_Q, which we call *child* and *ref* edges. Child edges are of type `child` and use the property `order` with positive integer values to encode the ordering; the subgraph of G_Q induced along `child` edges forms a tree. All other edges in E_Q are ref edges.

A symbolic token has:

- A kind, encoded as the property `kind`.
- A part of speech tag (from the Penn Treebank [23]), encoded as the property `partOfSpeech`.
- Optionally, additional kind-specific properties.
- Zero or more children, linked by `child` edges.

We use the notation {*kind/partOfSpeech* [*prop=value, ...*]: *child1, ..., childN*}, to refer to symbolic tokens, with unimportant parts omitted. For example, {LITERAL/NN} denotes a symbolic token of kind `LITERAL` with part of speech `NN` (singular noun). We also use the short-hand {"*e*"} to refer to a {LITERAL} token with an `expr` property *e*.

There is a token kind corresponding to each supported token type in the SQL AST, which has properties corresponding to the AST information, e.g. an SQL table reference has two string values for table name and table alias that the symbolic token also has. We will follow the convention of listing SQL AST tokens in upper camel case, e.g. {FromList}, and our extended tokens in all caps, e.g. {ROOT}. There are the following additional kinds, properties, and ref edge types:

- `ROOT`: The root of the entire tree.
- `LITERAL`: A natural language fragment.
  - `expr` property: The natural language expression.
- `AND/OR`: Conjunction by "and" or "or".
- `SEQ`: Sequence of symbols.

- `ATTR_LIST`: A list of ER-attributes.
- `IN_RELATIONSHIP`: Two entities participate in a relationship.
  - `e1`, `e2` edges: The two participating entity instances.
  - `relationship` edge: The ER diagram relationship.
- `ENTITY_INSTANCE`: An entity instance, corresponding to a table instance, perhaps restricted to a context.
  - `entity` edge: The ER diagram entity.
  - `singular` property: A singular label, defaulting to the singular label of `entity`.
  - `plural` property: A plural label, defaulting to the plural label of `entity`.
  - `cardinality` property: The cardinality of the instance, defaulting to `N`.
- `ENTITY_REF`: A reference to an entity instance.
  - `instance` edge: The referenced `{ENTITY_INSTANCE}`

These token kinds are sufficient for our subset of SQL.

4.3 Graph Construction

We construct G_{ER} directly from the ER diagram and G_Q as shown in Algorithm 1. We incorporate the ER-relational mapping M into G in two ways: first, we encode M_{ATTR} as $\xrightarrow{\texttt{attribute}}$ edges from G_Q to G_{ER}, and second, we encode the join mappings M_{FKJOIN} and M_{STJOIN} via properties of relationships in G_{ER}.

Algorithm 1 Initial Graph Construction.

$G_Q \leftarrow$ isomorphic to SQL AST
for table node t in `FromList` mapped to entity e **do**
  add new `{ENTITY_INSTANCE}` token s to G_Q
  add $s \xrightarrow{\texttt{entity}} e$ to G_Q
  for column reference $c \in G_Q$ referencing t **do**
    add $c \xrightarrow{\texttt{instance}} s$ to G_Q
  end for
end for
for col. reference $c \in G_Q$ mapped to ER-attribute n **do**
  add $c \xrightarrow{\texttt{attribute}} n$ to G_Q
end for
for relationship $rel \in G_{ER}$ **do**
  encode mapping as properties of rel
end for

Figure 3: Initial graph construction algorithm.

4.4 Rules

R is a set of rules where each rule $r \in R$ is defined by a predicate-action pair (p, A), denoted $r.\boldsymbol{p}$ and $r.\boldsymbol{A}$. A predicate p is a boolean formula of constraints over graph nodes, edges, and properties. There is a graph-matching function $\gamma : (G, p) \mapsto \{G' | G' \subseteq G \wedge G' \text{ satisfies } p\}$ that returns all subgraphs of G that satisfy the constraints of p. A rule action $A : G \mapsto \vec{a}$ maps a graph to an ordered sequence of actions $\vec{a}$, where each action $a \in \vec{a}$ is one of `{add_edge, delete_edge, add_node, delete_node, set_property}`.

We repeatedly apply rules as shown in Algorithm 2. Given the current graph G and the set of rules R, if $\exists r \in R$ such that $\exists G' \in \gamma(G, r.\boldsymbol{p})$, then execute $r.\boldsymbol{A}(G')$. If the result of executing $r.\boldsymbol{A}(G')$ is a new graph $\hat{G}$ not seen during evaluation, then update G, otherwise allow a different rule to apply. The goal of the rules is to transform G_Q from a structure resembling the SQL AST to a structure representing an output sentence.

The output condition is as follows. Consider the tree H of symbolic tokens induced along `child` edges of G_Q. If every leaf node in H is a `{LITERAL}` and all non-leaf nodes in

H are either {AND}, {OR}, {SEQ} or {ROOT} tokens, then we can read off a natural language sentence. {LITERAL} tokens hold the actual natural language expression used. {AND} and {OR} tokens have their children joined according to the rules of English. {SEQ} and {ROOT} tokens have their children joined with or without spaces depending on the rules of speech (e.g. spaces between two words but not a word and a comma).

Algorithm 2 Rule Application.

procedure EVALUATE-RULES(G, R)
  repeat
    $applied \leftarrow$ `false`
    for $r \in R$ s.t. $\exists G' \in \gamma(G, r.\boldsymbol{p})$ **do**
      $\vec{a} \leftarrow r.\boldsymbol{A}(G')$
      $\hat{G} \leftarrow$ result of applying $\vec{a}$ to G
      if $\hat{G} \neq G$ **then**
        $G \leftarrow \hat{G}$
        $applied \leftarrow$ `true`
      end if
    end for
  until not $applied$
  CHECK-OUTPUT(G)
end procedure

Figure 4: Rule application algorithm.

CHAPTER V: CONCRETE RULES

Rules vary in complexity, but fall into the following major groups: *a*) *analytic* rules discover some higher-level information by analysis of the state of G, *b*) *lowering* rules[3] produce {LITERAL} tokens from other symbolic token kinds, thus moving us towards an English output, and *c*) *grammar* rules work to reduce verbosity or otherwise smooth out the natural language output according to the rules of English. Here we focus on some interesting rules in our system; the remaining rules are listed in Section 2.

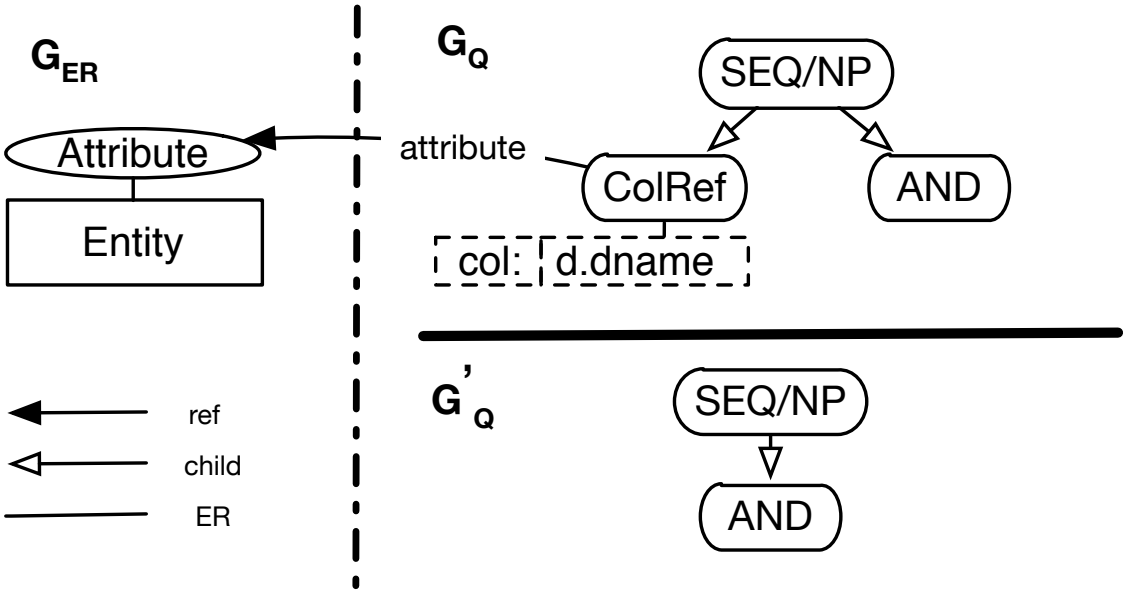

Figure 5: Graph transformation example.

First, let us define some low-level predicates that will be used to construct the rule predicates:

- isEdge(e, n_1, n_2): There is a graph edge e between graph nodes n_1 and n_2.
- nodeProp(n, p, v): There is a node n with property p having value v.
- edgeProp(e, p, v): There is an edge e with property p having value v.

[3]Lowering rules are so named by analogy to compilers that lower an IR into machine-specific code.

We use "_" to denote a variable we do not care about the identity of, where multiple occurrences of "_" are not necessarily equal to each other. We can construct additional predicates using these, e.g. the type of a node is

$$\text{nodeType}(n,t) \equiv \text{nodeProp}(n,\texttt{node-type},t)$$

the kind of a symbolic token is

$$\begin{aligned}\text{kind}(n,k) \equiv\ & \text{nodeType}(n,\texttt{symbolic}) \wedge \\ & \text{nodeProp}(n,\texttt{kind},k)\end{aligned}$$

and a ref edge with type ℓ is

$$\begin{aligned}\text{refs}(e,n_1,n_2,\ell) \equiv\ & \text{isEdge}(e,n_1,n_2) \wedge \\ & \text{edgeProp}(e,\texttt{edge-type},\ell) \wedge \\ & \ell \notin \Sigma_{L_{ER}} \cup \{\texttt{child}\}\end{aligned}$$

Our rules also make use of conjunct scopes or *c-scopes*, where a c-scope is either a single operator expression or a nesting of {AND} tokens with a common root, which ensures that the operators always hold together. For example, in the SQL expression "o_1 AND o_2", o_1 and o_2 occur in the same conjunct-scope, but in the expression "o_1 OR o_2" they occur in separate conjunct-scopes. For simplicity, we focus here on cases with a single c-scope, but more detailed discussion of c-scopes is found in Section 3.1.

5.1 Rule Application on a Running Example

To illustrate the application of each rule in the following sections, we will use the running example

(12) `SELECT fname, lname, address FROM`

```
employee AS e, department AS d WHERE
d.dname = 'Research' AND d.dnumber = e.dno
```

and a graphical presentation as shown in Figure 5. ER diagram elements use their standard shapes to represent different node types [12], while symbolic nodes use rounded-rectangles and are labeled with the token's kind and part of speech, if relevant. Properties are shown in a dashed box connected to the node or edge by a dashed edge. Child edges have unfilled arrow heads, while ref edges have filled arrow heads and are labeled with the edge type. ER diagram elements are read-only and shown to the left of a thick, dashed-and-dotted line. The subgraph matched by some $r.\boldsymbol{p}$ is shown above a thick, solid line and the result of applying $r.\boldsymbol{A}$ is shown below it. The initial state of G_Q for Query 12 is shown in Figures 5 and 6.

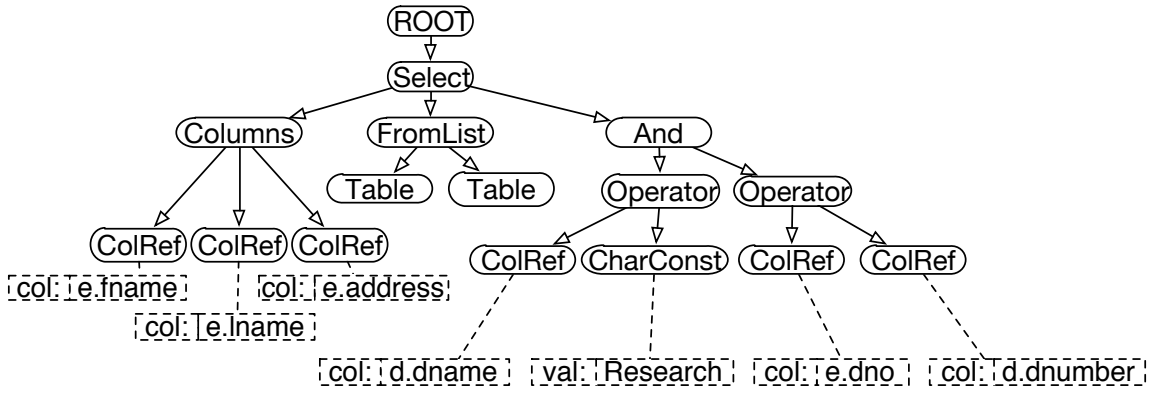

Figure 6: Subgraph of initial state isomorphic to SQL AST.

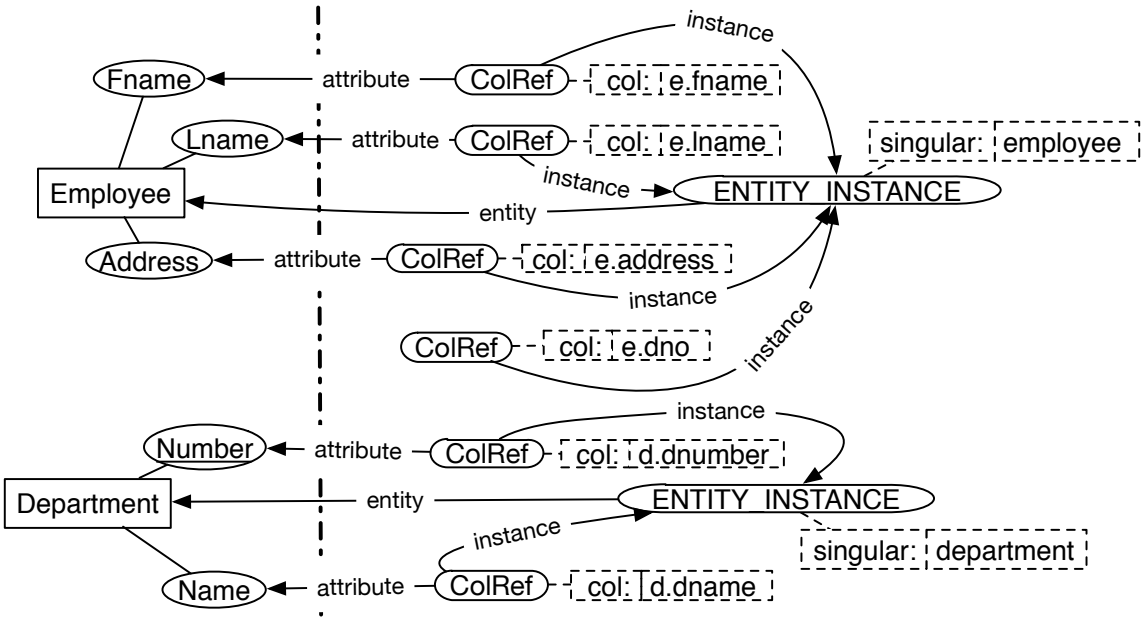

Figure 7: Ref edges of initial state.

5.1.1 Describing Attribute Rule

The Describing Attribute Rule is an analytic rule that uses a literal value to describe an entity instance. The predicate matches an expression $c = v$, where c is a column mapped to an ER-attribute a annotated as a describing attribute and v is a string constant, which the rule consumes and uses to relabel the associated entity instance. The Describing Attribute Rule's predicate is:

$$
\begin{aligned}
r.\boldsymbol{p} \equiv & \exists a, t, b, c, n_v, e_a, p, n_{inst}, s_1, s_2 \\
& \text{descAttr}(a, t) \wedge \text{refs}(e_a, c, a, \texttt{attribute}) \\
& \wedge \text{compare}(b, c, n_v, \text{`='}) \wedge \text{kind}(n_v, \texttt{CharConst}) \\
& \wedge \text{isChild}(p, b, \_) \wedge \text{nodeProp}(n_v, \texttt{value}, v) \\
& \wedge \text{refs}(e_{inst}, c, n_{inst}, \texttt{instance}) \\
& \wedge \text{cscopeOf}(n_{inst}, s_1) \wedge \text{inCScope}(b, s_2)
\end{aligned}
$$

The action $r.\boldsymbol{A}$ is defined as follows. We focus here on the case where the comparison b and {ENTITY_INSTANCE} token n_{inst} are in the same conjunct-scope, $s_1 = s_2$, although this rule also handles the case where $s_1 \neq s_2$. First, delete the comparison b, as in delete_node(b). Let *card* be 1 if a is a key attribute and N otherwise, then update the cardinality with *card* and the labels with a function of v depending on the describing-type t, e.g. if $t =$ prepend and the previous singular label is x, then the new singular label takes the form "v x."

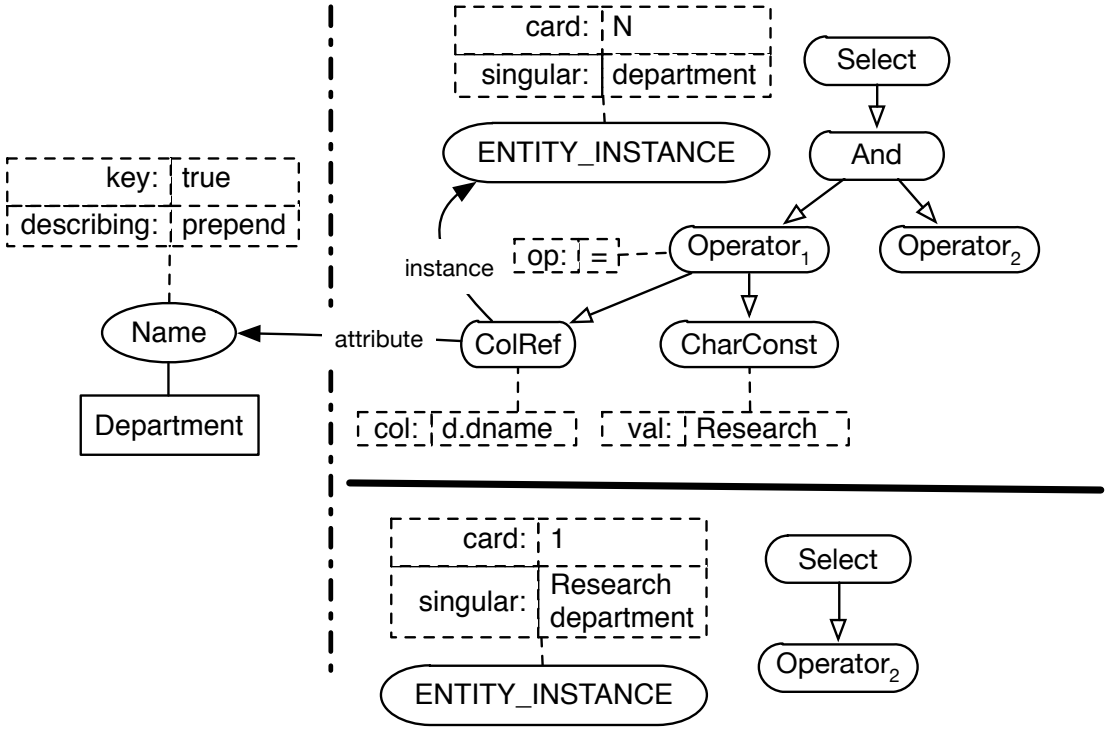

Figure 8: Describing Attribute Rule Application.

Figure 8 shows the result of applying the Describing Attribute Rule to our running example. In this case, the department entity instance n_{inst} is relabeled to "Research department" and the cardinality is set to 1. The {Operator} node b with the comparison is deleted and its sibling node takes the place of the {AND} parent p.

5.1.2 Join Relationship Rule

A major goal of the rules is to incorporate information by analyzing the query and use this in place of naive verbalization. The Join Relationship Rule is an analytic rule that detects table joins that are mapped to a relationship in the ER diagram. The foreign-key join type[4] predicate is:

$$
\begin{aligned}
r.\boldsymbol{p} \equiv \exists & rel, tn_1, cn_1, tn_2, cn_2, c_1, c_2, s \\
& \text{erFkJoin}(rel, tn_1, cn_1, tn_2, cn_2) \wedge \text{cscope}(s) \\
& \wedge \text{inCScope}(c_1, s) \wedge \text{inCScope}(c_2, s) \\
& \wedge \text{colCompare}(c_1, c_2, \text{`='}) \\
& \wedge \text{colTableName}(c_1, tn_1) \\
& \wedge \text{nodeProp}(c_1, \texttt{name}, cn_1) \\
& \wedge \text{colTableName}(c_2, tn_2) \\
& \wedge \text{nodeProp}(c_2, \texttt{name}, cn_2)
\end{aligned}
$$

This predicate says that there is a foreign-key join in the ER-relational mapping for relationship *rel* defined by keys $tn_1.cn_1$ and $tn_2.cn_2$ with an equality comparison between these keys in the same conjunct-scope s. The colTableName predicate resolves table aliasing in the column references.

The Join Relationship Rule action for the foreign-key join is defined as follows. Replace the first column comparison token with a new `{IN_RELATIONSHIP}` token, with $\xrightarrow{\texttt{e1}}$ and $\xrightarrow{\texttt{e2}}$ edges pointing to the respective `{ENTITY_INSTANCE}` tokens of the replaced `{ColRef}` tokens, and the $\xrightarrow{\texttt{rel}}$ edge pointing to the `relationship` node of *rel* in G_{ER}. These oper-

[4]The separate-table join type is not shown due to space considerations.

ations are returned as a sequence of low-level graph rewriting operations as defined in Chapter IV, Section 4.4.

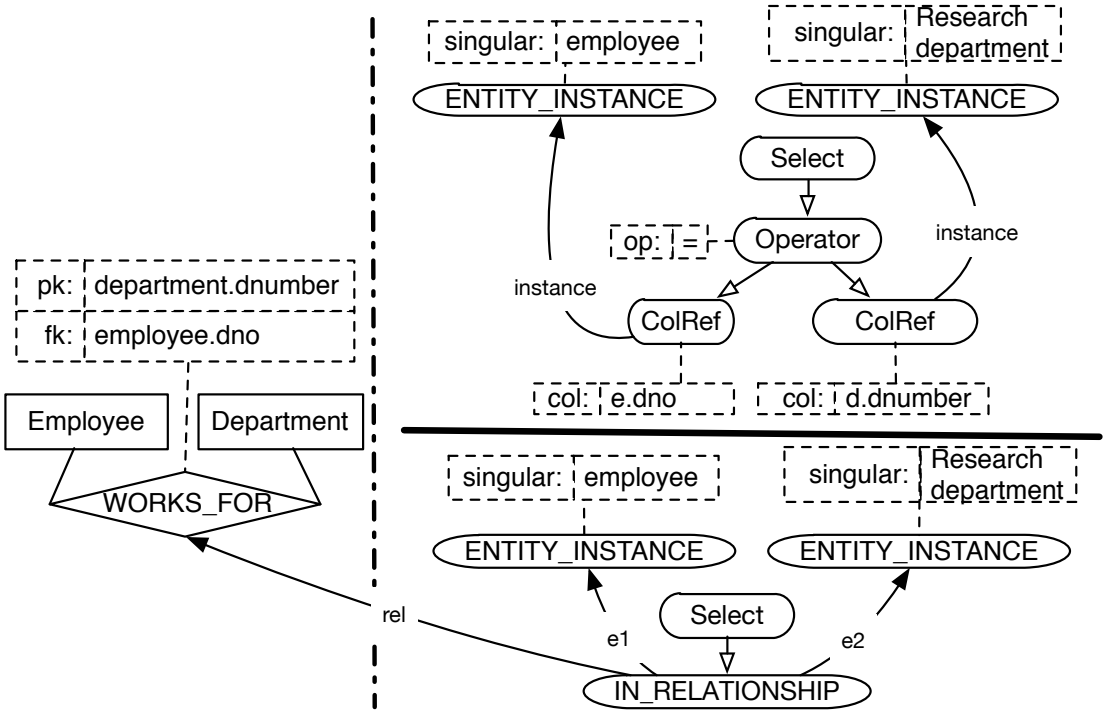

Figure 9: Join Relationship Rule Application.

The purpose of the Join Relationship Rule's rewrite action is to replace the low-level join conditions with their high-level meaning, which is participation in a relationship. Figure 9 shows the result of applying this rule to our example, where the `{Operator}` token is replaced by an `{IN_RELATIONSHIP}` token which captures the fact the employee entity instance works for the department entity instance.

5.1.3 In Relationship Lowering Rule

The In Relationship Lowering Rule is a lowering rule that converts an `{IN_RELATIONSHIP}` token to a partial verbalization of the relationship it reflects. *r.**p*** simply matches an `{IN_RELATIONSHIP}` token along with the relationship and two `{ENTITY_INSTANCE}` tokens it references. *r.**A*** replaces the `{IN_RELATIONSHIP}` token with a verb phrase (`{SEQ/VP}`)

which is an {ENTITY_REF} pointing to the first {ENTITY_INSTANCE}, one or more {LITERAL} tokens verbalizing the relationship, and an {ENTITY_REF} pointing to the second {ENTITY_INSTANCE}.

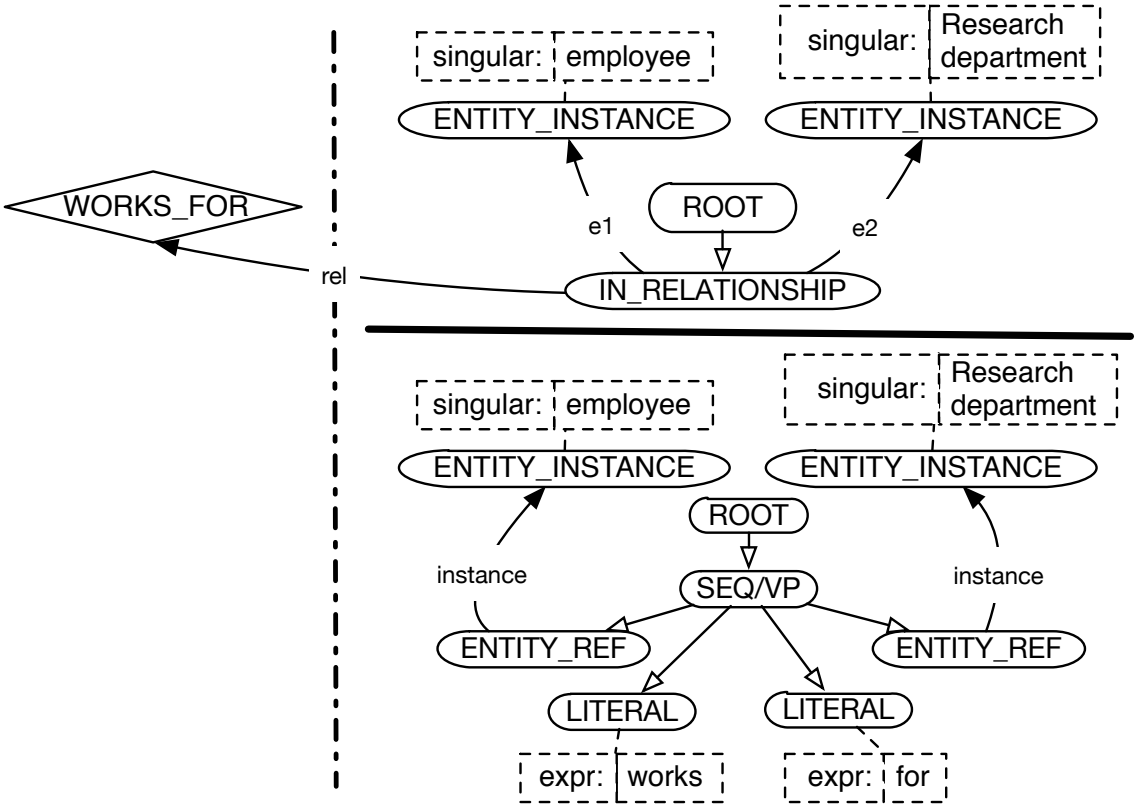

Figure 10: In Relationship Lowering Rule Application.

Figure 10 shows the result of applying this rule to our running example. Note that we have skipped the application of several rules, but the {IN_RELATIONSHIP} token is the same token introduced by the Join Relationship Rule in Chapter V, Section 1.2. The result of this application is to introduce a verb phrase expressing the relationship. The two entity instances are left as symbolic tokens, while the relationship is lowered into the expression "works for" as {LITERAL} tokens.

5.1.4 Single Reference Anaphora Rule

The Single Reference Anaphora Rule is a grammar rule that attempts to use an anaphoric

reference [11] such as “who” or “whose” to express either verb phrases or possessive references, respectively; we focus on the former case here. The predicate $r.\boldsymbol{p}$ is:

$$
\begin{aligned}
r.\boldsymbol{p} \equiv & \exists r_1, w, n_1, w, i_w, e \\
& \text{kind}(r_1, \texttt{ENTITY\_REF}) \wedge \text{kind}(w, \texttt{WHERE}) \wedge r_1 < w \\
& \wedge \text{isChild}(n_1, w, i_w) \wedge \text{nodeProp}(r_1, \texttt{instance}, e) \\
& \wedge \forall r_2[(\text{kind}(r_2, \texttt{ENTITY\_REF}) \wedge r_2 < w) \implies r_2 = r_1] \\
& \wedge \forall t[\text{nodeProp}(t, \texttt{partOfSpeech}, p) \wedge \neg \text{isPronoun}(p)] \\
& \wedge \forall s[(\text{isChild}(n_1, s, i_s) \wedge i_s > i_w) \implies \\
& (\text{nodeProp}(s, \texttt{partOfSpeech}, \texttt{VP}) \wedge \text{isChild}(s, n_2, 0) \\
& \wedge \text{kind}(r_3, \texttt{ENTITY\_REF}) \wedge \text{nodeProp}(r_3, \texttt{instance}, e))]
\end{aligned}
$$

$r.\boldsymbol{p}$ matches the following condition: *i)* there is a single `{ENTITY_REF}` token referencing an `{ENTITY_INSTANCE}` token e that occurs before a `{WHERE}` token w, *ii)* there is no other `{ENTITY_REF}` token before w, *iii)* every sibling of w occurring after it is a verb phrase s, *iv)* the first child of each such s is an `{ENTITY_REF}` token r_3 referencing e, and *v)* there is no existing pronoun use. $r.\boldsymbol{A}$ is defined to replace w with a `{"who"}` token if e's `entity-type` is `person` or `{"that"}` otherwise, and delete each token r_3.

Figure 11 shows the result of applying this rule to our running example. In this case, the employee instance is used as the base of the anaphoric reference. The tokens created by the In Relationship Lowering Rule are simplified by deleting the leading `{ENTITY_REF}` token.

5.1.5 Determiner Redundancy Rule

Finally, some rules exist to smooth out the natural language structure. The Determiner Redundancy Rule handles cases where a determiner (e.g. “the” or “a”) is redundant and

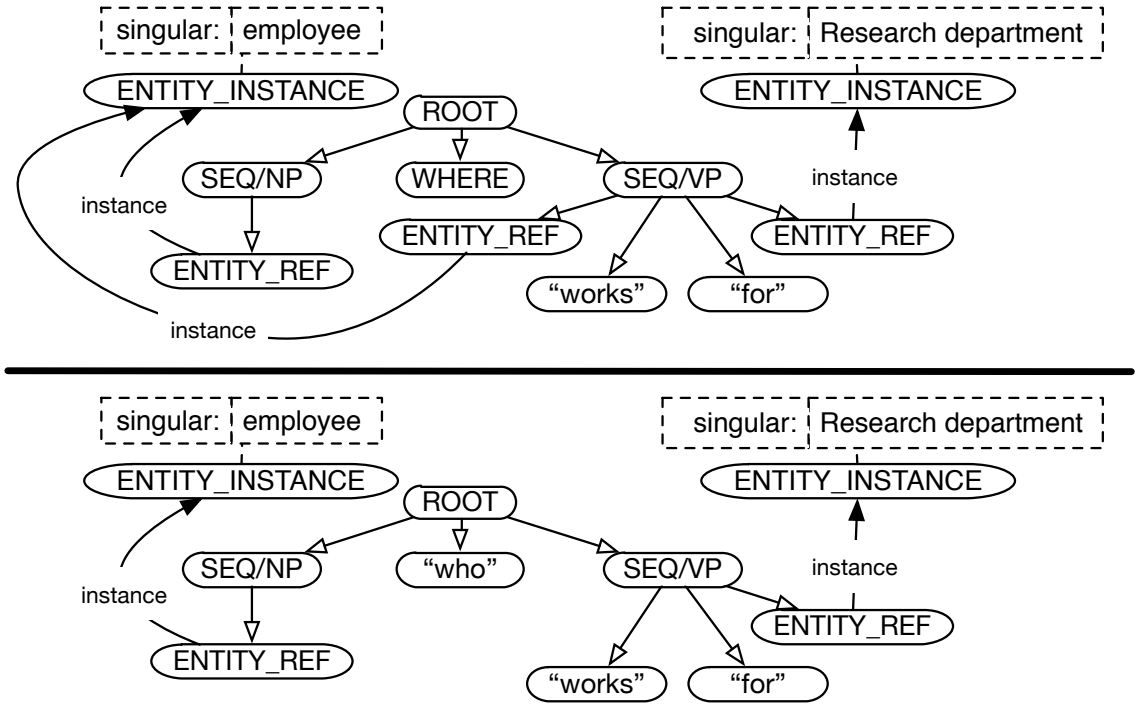

Figure 11: Single Reference Anaphora Rule Application.

can be removed to make the natural language less verbose. The predicate is defined as:

$$
\begin{aligned}
r.\boldsymbol{p} \equiv & \exists p,k,n_1,c_1,d_1,e \\
& \text{kind}(p,k) \wedge k \in \{\texttt{AND},\texttt{OR},\texttt{ATTR\_LIST}\} \\
& \wedge \text{isChild}(p,n_1,c_1) \wedge \text{kind}(n_1,\texttt{SEQ}) \\
& \wedge \text{isChild}(n_1,d_1,0) \wedge \text{isDet}(d_1,e) \\
& \forall n_2,d_2,c_2[(\text{isChild}(p,n_2,c_2) \wedge c_2 > c_1) \implies \\
& \quad (\text{kind}(n_2,\texttt{SEQ}) \wedge \text{isChild}(n_2,d_2,0) \\
& \quad \wedge \text{isDet}(d_2,e))]
\end{aligned}
$$

This predicate says there is an {AND}, {OR}, or {ATTR_LIST} token with two or more {SEQ} children, each of which has a {LITERAL/DT} first child and all of the {LITERAL/DT} tokens have the same expression e. At a high-level, the Determiner Redundancy Rule's

action $r.\boldsymbol{A}$ is simply to delete all d_2 tokens. In natural language, this transformation is equivalent to replacing "the a, the b, and the c" with "the a, b, and c."

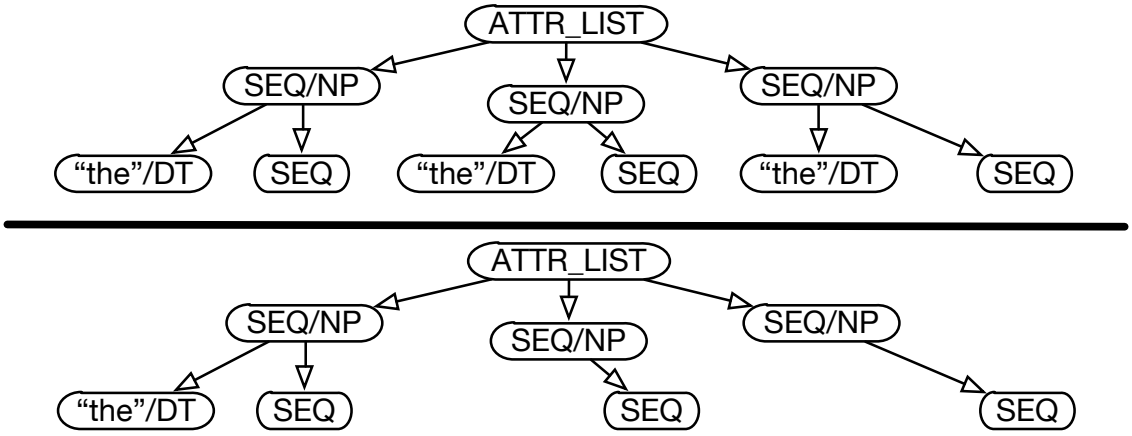

Figure 12: Determiner Redundancy Rule Application.

This is the final rule to fire for our running example. After a number of simple lowering rules were applied, which we've omitted the details of, we were in a potential output state that could be read as natural language. However, even in this case some improvement is still possible. Figure 12 shows the result of applying this rule in the penultimate state. After this application, we have the state shown in Figure 13, yielding the final output for Query 12:

(12) Select the first name, last name and address of each employee who works for the Research department.

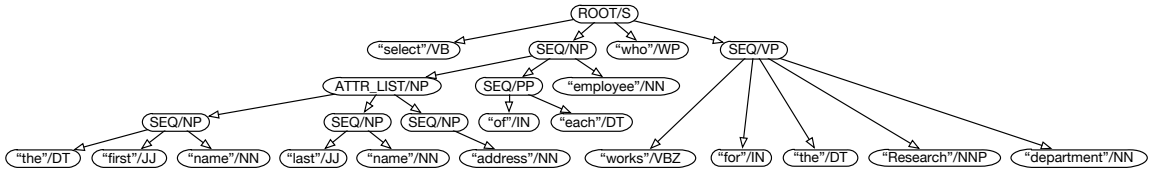

Figure 13: Final symbolic state of Section 1.

5.2 Additional Rules

In this section, the remaining rules that have not been explained thus far will be presented.

5.2.1 Basic Lowering Rules

A number of rules simply take a token of a certain kind and convert it to a literal value. These rules follow the general template of having a predicate $r.\boldsymbol{p} \equiv \text{kind}(n,k) \wedge \text{nodeProp}(n, \texttt{partOfSpeech}, p)$ for a particular token kind k and defining $r.\boldsymbol{A}$ to replace n with a new $\{\texttt{LITERAL}/p\}$ token n', where the `expr` attribute of n' is defined separately for each rule.[5]

The basic lowering rules are as follows:

- Select Label Rule: $k = \texttt{SELECT}$ and the `expr` of n' is "select" or a synonym.
- Attribute Literal Label Rule: k is `ATTRIBUTE` or `ColRef` and n maps to an ER-attribute, and the `expr` of n' is the ER-attribute's singular or plural label.
- Column Reference Literal Rule: $k = \texttt{ColRef}$ where n is not mapped to any attribute and the `expr` of n' verbalizes the column name.
- Entity Ref Lowering Rule: $k = \texttt{ENTITY\_REF}$ and the `expr` of n' is the singular or plural label of n, depending on whether p is singular or plural.
- Where Literal Rule: $k = \texttt{WHERE}$ and the `expr` of n' is "such that" or a synonym.
- Binary Comparison Rule: $k = \texttt{Operator}$ and n' is a `{SEQ}` of `{LITERAL}` tokens. Let the two children of n be c_1 and c_2, then the children of n' take the form $c_1, e_1, ..., e_n, c_2$, where $e = e_1, ..., e_n$ verbalizes the operator of n, e.g. if the operator of n is ">", then e is "is greater than" and similarly for the other comparison operators. The type of

[5] Multi-word expressions actually use a `{SEQ}` of `{LITERAL}` tokens.

c_1 and c_2 is also used to determine the verbalization, e.g. to use "later than" for dates and times.

- Number Literal Rule: $k =$ `NUMBER`, $p =$ `CD` and the `expr` of n' is the `value` v of n formatted according to the `data-type` dt of n, e.g. if $dt =$ `dollars` and $v = 100$, then "$100" is the `expr` of n'.

- Between Literal Rule: $k =$ `Between` with three children c_1, c_2, c_3, where c_1 is a column reference and c_2, c_3 are numbers, and n' is a `{SEQ}` with children of the form "c_1 is between c_2 and c_3".

- All Attributes Literal Label Rule: $k =$ `ALL_ATTRIBUTES` and the `expr` of n' is "all attributes".

- Default Is Null Rule: $k =$ `IsNull` and the `expr` is "exists" or "does not exist" appended to the child token.

5.2.2 Select List Format Rule

The Select List Format Rule verbalizes the attribute lists from the query. The rule predicate is:

$$
\begin{aligned}
r.\boldsymbol{p} \equiv & \exists s, f, \mathit{fl} \\
& \text{kind}(s, \texttt{SelectList}) \wedge \text{kind}(f, \texttt{From}) \\
& \wedge \text{kind}(\mathit{fl}, \texttt{FromList}) \wedge \\
& \forall t [\text{kind}(t, \texttt{Table}) \wedge \text{kind}(e, \texttt{ENTITY\_INSTANCE}) \wedge \\
& \forall c [\text{kind}(c, \texttt{ColRef}) \wedge \text{refs}(\_, c, e, \texttt{instance}) \wedge \\
& \text{refs}(\_, c, t, \texttt{table}) \wedge \text{refs}(\_, c, a, \texttt{attribute}) \\
& \wedge \text{isChild}(s, c, \_)]]
\end{aligned}
$$

This predicate matches all columns selected in the query, grouped by each table in the from-list portion of the query.

For $r.\boldsymbol{A}$, consider first each $c_{i,j}$ associated with some table t_i with corresponding entity instance e_i. $r.\boldsymbol{A}$ will produce a structure which looks like the following: `{SEQ: {AND:` $c_{i,1}$ `, ... ,` $c_{i,m}$`}, {"of"}, {"`x`"},` e_i `}`, where x is "each" or "all" if e_i has a singular or plural part of speech, respectively. Let n be the `{SEQ}` if there is only one such t, or a new `{AND}` token where the children are all the `{SEQ}` tokens if there are are multiple such t. Then $r.\boldsymbol{A}$ replaces the select list s with n and deletes the `{From}` clause f.

5.2.3 Analytic Rules

- Value Type Inference Rule: The Value Type Inference Rule propagates the data type of an ER-attribute to constant values that are compared with it. This rule matches `{Between}` and `{Operator}` tokens, where one child is a `{ColRef}` mapped to an ER-attribute a and the other is a value token (e.g. `{CharConst}`) v, where the data

types of a and v differ. The action of this rule sets the data type of v to that of a. In Example 4-10, the Value Type Inference Rule learns that the expression `20000` is in dollars by inference from the data type of the `Salary` ER-attribute.

- Range To Between Rule: The Range To Between Rule converts sets of `{Operator}` tokens that imply a bound on a particular `{ColRef}` to a `{Between}` token; this is equivalent to converting the SQL expression `c >= 10 AND c <= 20` to `c BETWEEN 10 AND 20`.

- Merge Composite Attribute Rule: The Merge Composite Attribute Rule collapses a set of `{ATTRIBUTE}` tokens (or `{ColRef}`s mapping to ER-attributes) that are children of a composite ER-attribute into a single `{ATTRIBUTE}` token mapped to the composite ER-attribute. For example, for the company database, if `fname`, `minit` and `lname` appear in the SQL column list, they will be replaced an `{ATTRIBUTE}` token mapping to the `Employee.Name` ER-attribute.

- Implicit Relationship Rule: The Implicit Relationship Rule detects relationships with implicit entities that are not directly referenced in the query. Specifically, this rule matches comparisons of a foreign key column with a direct value, e.g. `{CharConst}` or `{IsNull}`. The rule's action creates a new `{ENTITY_INSTANCE}` for the implicit entity, inserts a key-join condition (which will be matched by the Join Relationship Rule), and changes the value comparison token to use the primary key of the implicit entity. Later, in Example 1-6, this rule introduces the "supervisor" entity.

- Not In Relationship Rule: The Not In Relationship Rule verbalizes participation in a relationship where one of the entities does not exist. This rule matches an `{IN_RELATIONSHIP}` token with a corresponding `{IsNull}` token constraining the primary key of one of the participating entity instances. The rule consumes these tokens and verbalizes the negative relationship. Later, in Example 1-6, the Not In Relationship Rule is responsible for the expression "does not have any supervisor."

5.2.4 Grammar Rules

- In Relationship Label Rule: The In Relationship Label Rule uses relationship role labels to relabel entity instances that participate in a relationship. This rule matches an {IN_RELATIONSHIP} token that maps to a relationship *r* that has role labels for the two {ENTITY_INSTANCE} tokens. The rule consumes the {IN_RELATIONSHIP} token and updates the labels of one of the {ENTITY_INSTANCE} tokens, depending on the current label and order of associated {ENTITY_REF} tokens. Later, the use of "supervisor" in Example 1-8 is due to this rule.

- Anaphoric Reference Rule: The Anaphoric Reference Rule converts references to an entity instance to anaphoric references, i.e. using pronouns for all but the first. This rule looks for an {ENTITY_REF} token to use as the base and a series of {ENTITY_REF} tokens for the same entity instance that are not blocked by, for example, references to another entity, being possessed by another reference, or by existing use of pronouns. All the matching {ENTITY_REF} tokens other than the base are converted to pronoun form. Later, in Example 1-8, this rule introduces the term "their."

- Entity Ref Needs Id: The Entity Ref Needs Id Rule prevents the ambiguous labeling of different entity instances. This rule matches two or more {ENTITY_INSTANCE} tokens with the same label that have some respectively associated {ENTITY_REF} tokens. This rule adds a unique variable to each label so that the references will not be ambiguous when they are lowered to {LITERAL} tokens.

- Simplify Repeated Attributes Rule: The Simplify Repeated Attributes Rule collapses attribute list expressions where the phrasing is repeated. For example, in "name of each *x* and name of each *y*", the attribute expressions ("name") and prepositional

phrases ("of each") match, so this rule condenses the full expression into the equivalent of "name of each x and y."

5.3 Safety and Conjunct Scopes

For some simple rules, such as the Value Type Inference Rule from Section 2.1, it is clear that no information is being lost. However, rules which conduct more significant updates must be careful about how they manage information. This section describes conjunct-scopes and their role in rule safety.

5.3.1 Conjunct Scopes

A conjunct-scope or *c-scope* is associated with a node in the query graph G_Q and defines the range of influence of conditions in the `WHERE` clause of the SQL query. For a node n and c-scope s, $s = n$ if n is the `{ROOT}` token or the direct child of an `{OR}` token (isChild$(n', n, _) \wedge$ kind$(n', \texttt{OR})$), or the c-scope of n's parent otherwise. For any two constraint tokens $n_1 \neq n_2$, if cscope$(n_1, s) \wedge$ cscope(n_2, s) then n_1 and n_2 always apply together when the query is executed.

Let s_0 denote the conjunct-scope of the `{ROOT}` token. s_0 represents the global scope of the entire query. At each disjunction (`{OR}`) t, there is a new conjunct-scope s' for each child token. So, for all tokens $n \in G_Q$ that are connected by child edges, there is a conjunct-scope defined to be either one of the s' c-scopes or s_0.

Next, consider the `{ENTITY_INSTANCE}` tokens, which exist without being connected by any `child` edges in G_Q. We define a partial order $\leq$ on each c-scope s such that $s_i \leq s_j$ if $|s_i| \leq |s_j|$ where $|s|$ denotes the depth of s in the tree induced along `child` edges in G_Q. For an `{ENTITY_INSTANCE}` token n, let $S(n)$ denote the set of conjunct-scopes of incoming `instance` references $\{s | \text{cscope}(n', s) \wedge \text{refs}(n', n, \texttt{instance})\}$, then the c-scope of n is $\min S(n)$.

5.3.2 Safety Examples

Consider the Describing Attribute Rule from Section 1.1. The Describing Attribute Rule discovers a new identity for some entity instance and relabels it as such. In some cases, this rule will update an existing `{ENTITY_INSTANCE}`; for this to be safe, we require that the change not be visible outside of the scope where the constraint occurs. Consider the query:

(13)
```
SELECT d.location FROM department d
    WHERE d.name = 'R' OR d.name = 'S';
```

Let c_1 be the `d.location` token, c_2 and c_3 be the first and second `d.name` tokens, respectively, and b_1 and b_2 be the first and second equality operators, respectively. There are three c-scopes s_0, s_1, s_2 in this query, where s_0 is the global c-scope, $s_1 = b_1$ and $s_2 = b_2$ (by virtue of being children of an `{OR}` token.) Additionally, there is a `{ENTITY_INSTANCE}` token, n_{inst}, with an incoming `instance` edge from each c_i. If n_{inst} were to be updated based on `d.name = 'R'`, then the identity implied by the department name being "R" would apply to `d.location` as well. However, this query may also return department locations where the name is not "R". The c-scope of n_{inst} is s_0 due to the incoming edge from c_1 (`d.location`), therefore this update is prohibited. The solution in this case is to create a new `{ENTITY_INSTANCE}` for the c-scope and introduce an `{IS}` token referencing it that will express the identity change.

Contrast this with the query:

(14)
```
SELECT d.location FROM department d
    WHERE d.name = 'R';
```

In this case, there is only the global c-scope s_0. n_{inst} may be updated because all `d.location` tuples are under the constraint `d.name = 'R'`.

CHAPTER VI: IMPLEMENTATION

We implemented our framework and rule set into a working system. We plan to make this system open-source and deploy it for use in educational settings.

In order to implement rule evaluation as described in Chapter IV, we require both a method of encoding each rule predicate $r.\boldsymbol{p}$ and a way to evaluate this predicate encoding against G for the γ function. We use Datalog with stratified negation [5] for both of these purposes. Each basic characteristic of G is represented as a separate Datalog relation; this includes nodes, edges, and property values. The current structure of G in any iteration is encoded as a set of Datalog facts under these relations. Higher-level predicates such as those shown in Chapter 5 are written as Datalog rules and evaluated to find matching subgraphs G' using the IRIS reasoner [6]. A matching subgraph $G' \in \gamma(G, r.\boldsymbol{p})$ is simply the union of all graph elements appearing in the resulting relations of one or more Datalog queries. Although we made this design decision independently, previous work on graph-based query languages has shown and exploited correspondence between graph matching and Datalog programs [10, 17], which justifies our approach.

Datalog predicates express universal quantification over the variables in the head and body literals, such that the resulting relation holds every matching tuple inferable by least-fixed-point iteration starting from the input facts. We evaluate existentially quantified variables in our rule predicates by whether or not the Datalog relation is empty. First, consider a predicate with only existential quantification, such as the Describing Attribute Rule predicate from Chapter V, Section 1.1. If the Datalog relation for this predicate is non-empty, then we use each tuple in the relation to produce the respective subgraphs $G' \in \gamma(G, r.\boldsymbol{p})$. So, we can evaluate all such matches using a single Datalog query.

Next, consider predicates with universal quantification, such as the Determiner Redundancy Rule from Chapter V, Section 1.5. The existential variables in the outermost scope

may still be evaluated in a single Datalog query using negation. Informally, the pattern is as follows. First, define a predicate which matches the outer scope along with a violation of the universally quantified inner scope. Next, define and evaluate a predicate which matches the outer scope and the negation of this violation-detecting predicate. Any tuples in this relation are such that the universally-quantified predicates hold. For example, if a rule's original predicate formula is $\exists x\, p_1(x) \wedge \forall y\, p_2(x,y)$, then we find a violation with $\phi_v(x,y) \equiv p_1(x) \wedge \neg p_2(x,y)$ and evaluate assignments for the original formula with $\phi(x) \equiv p_1(x) \wedge \neg\phi_v(x,y)$. Although we capture all the existential variables in a single Datalog query, we still need all instances of the universally-quantified variables to construct G'. We use one additional query to gather the match's instances relative to a particular tuple in the outer relation. The initial query for the outer scope populates many intermediate relations via fixed-point computation, so the runtime cost of these additional queries is greatly reduced by reusing these intermediate results and not recomputing them.[6] All of the translations in our evaluation (Chapter VII) are produced in less than two seconds on a laptop with a 2 GHz quad-core processor.

Finally, our implementation is deterministic and always produces a single translation for a given input, unlike our formal algorithm which allows various translations to be produced by different rule application orderings. In practice, we impose a partial ordering on the rules and apply them according to this ordering. For example, more sophisticated analytic rules apply before basic lowering rules. This ensures that we terminate quickly and output a high-quality translation. With non-deterministic rule ordering, basic lowering rules might be applied early, which can prevent some analytic rules from applying and lead to a translation that is more mechanical and verbose. Although it is outside the scope of this work, it should be possible to use other techniques to automatically determine rule orderings, e.g. by using Markov Logic Networks [27].

[6] This behavior depends on the Datalog engine's implementation, but is a common characteristic.

CHAPTER VII: EVALUATION

In this section, we evaluate our technique on different application scenarios such as those discussed in Chapter II. In addition, we evaluate the complexity of our system in terms of the rules needed to produce translations and the applicability of individual rules across all translations.

7.1 Textbook Translations

For the first experiment, we selected ten queries from a database education textbook [12]. The textbook contains an English description of each query written by the authors. The goal of this experiment is to see whether our technique is able to closely approximate the descriptions given by humans for these queries. In order to use our technique for providing feedback, the generated output must be clear, concise, and close to how a person would describe the query. Table 1 shows the ten queries for the Company database, the textbook description, and the description generated by our technique. We will use, e.g. Example 1-3 to refer to the third query in Table 1.

As one can see, many of the results are very close to the human-provided descriptions. In all cases, the translation produced by our system is an accurate reflection of the original query. Our system's translations tend to be more verbose than the human equivalent, although sometimes the opposite is true, as in Example 1-3. Examples 1-8 and 1-10 demonstrate how multiple rules interact to produce sophisticated translations. In Example 1-8, the In Relationship Label Rule uses the role label "supervisor" rather than expressing the "supervises" relationship, while the Simplify Repeated Attributes Rule compacts the common use of "last name" to produce the concise translation shown. In Example 1-10, the In Relationship Label Rule applies twice to obtain the "controlling department" and "manager" labels, the Simplify Repeated Attributes Rule compacts the uses of "number"

for the project and department, and the Anaphoric Pronoun Rule shortens a possessive reference to the project into "its"; the resulting translation is only slightly longer than the human-provided version.

Example 1-9 is another notable translation, where the Entity Ref Needs Id Rule has introduced IDs for the two employee entities to prevent the ambiguous use of “employee” to refer to both. A rule that learned an alternative way of referencing one of the two employees, such as “department’s manager” for *e2*, would make this disambiguation unnecessary and the Entity Ref Needs Id Rule would not be applied. Also note that the human-provided translation is inaccurate, as the query requests all attributes of all three tables, not just the first `employee` instance. Similarly, in Examples 1-5 and 1-6, the human-provided translations imprecisely refer to the `fname` and `lname` columns collectively as “name”, which is ambiguous since the `minit` column is also part of the employee name.

7.2 Problem Generation

The next experiment is motivated by problem generation and plagiarism prevention. In this scenario, an instructor has a set of problems in the form of queries and problem descriptions, written against a particular ER diagram and schema. The instructor may use these queries as templates and create new queries against a different ER diagram, while our approach will generate the descriptions for these new queries. Table 2 shows the results of this experiment. The second column is the original query for the Company database and the third contains the natural language generated for this query. The fourth and fifth columns show queries and their natural language translations, respectively, that are created for a database that models business trips using the query in the first column as a template.

As the results show, the natural language output of our system is structurally similar when the inputs follow the same syntactic structure, but with key differences. In Examples 2-14 and 2-15, the Join Relationship Rule has suppressed the low-level equalities and instead caused the semantic relationships to be expressed. In Example 2-10, the greater-than operator (>) and the constant values below it are expressed differently because the data

Table 1: Textbook translations experiment.

ID	Query	Textbook NL	Generated NL
1	`SELECT ALL salary FROM employee`	Retrieve the salary of every employee.	Select the salary of each employee.
2	`SELECT DISTINCT salary FROM employee`	Retrieve all distinct salary values.	Select the distinct salaries of all employees.
3	`SELECT ssn FROM employee`	Select all employee ssns in the database.	Select the ssn of each employee.
4	`SELECT * FROM employee WHERE (salary BETWEEN 30000 AND 40000) AND dno = 5`	Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.	Select all attributes of each employee who works for department 5 and whose salary is between $30,000 and $40,000.
5	`SELECT fname, lname, address FROM employee, department WHERE dname = 'Research' AND dnumber = dno`	Retrieve the name and address of all employees who work for the "Research" department.	Select the first name, last name, and address of each employee who works for the Research department.
6	`SELECT fname, lname FROM employee WHERE super_ssn IS NULL`	Retrieve the names of all employees who do not have supervisors.	Select the first name and last name of each employee who does not have any supervisor.
7	`SELECT bdate, address FROM employee WHERE fname = 'John' AND minit = 'B' AND lname = 'Smith'`	Retrieve the birth date and address of the employee(s) whose name is "John B. Smith".	Select the birth date and address of each employee whose first name is "John", middle initial is "B", and last name is "Smith."
8	`SELECT E.lname AS employee_name, S.lname AS supervisor_name FROM employee AS E, employee AS S WHERE E.super_ssn = S.ssn`	Retrieve the last name of each employee and his or her supervisor.	Select the last names of each employee and their supervisor.
9	`SELECT * FROM employee E, employee M, department D WHERE E.salary > M.salary AND E.dno = D.dnumber AND D.mgr_ssn = M.ssn`	Retrieve the employees whose salary is greater than the salary of the manager of the department that the employee works for.	Select all attributes of each employee *e*, employee *e2*, and department where *e*'s salary is greater than *e2*'s salary, *e* works for the department, and *e2* manages the department.
10	`SELECT pnumber, dnumber, lname, address, bdate FROM project p, department d, employee e WHERE p.dnum = d.dnumber AND d.mgr_ssn = e.ssn AND p.plocation = 'Stafford'`	For every project located in "Stafford", list the project number, the controlling department number, and the department manager's last name, address, and birthdate.	Select the numbers of each project and its controlling department and the last name, address, and birth date of each controlling department's manager where the project's location is "Stafford."

Table 2: Problem generation experiment.

	Original Query	Original NL	Query Variant	Variant NL
1	`SELECT ssn FROM employee`	Select the ssn of each employee.	`SELECT name FROM salesperson`	Select the name of each salesperson.
2	`SELECT ssn, salary FROM employee`	Select the ssn and salary of each employee.	`SELECT name, dept_no FROM salesperson`	Select the name and department number of each salesperson.
3	`SELECT * FROM employee`	Select all attributes of each employee.	`SELECT * FROM salesperson`	Select all attributes of each salesperson.
4	`SELECT DISTINCT fname FROM employee`	Select the distinct first names of all employees.	`SELECT DISTINCT name FROM salesperson`	Select the distinct names of all salespersons.
5	`SELECT DISTINCT fname, salary FROM employee`	Select the distinct values of the first names and salaries of all employees.	`SELECT DISTINCT name, dept_no FROM salesperson`	Select the distinct values of the names and department numbers of all salespersons.
6	`SELECT DISTINCT * FROM employee`	Select the distinct values of all attributes of all employees.	`SELECT DISTINCT * FROM salesperson`	Select the distinct values of all attributes of all salespersons.
7	`SELECT ssn FROM employee WHERE fname = 'Ahmad'`	Select the ssn of each employee whose first name is "Ahmad."	`SELECT ssn FROM salesperson WHERE name = 'John Smith'`	Select the ssn of each salesperson whose name is "John Smith."
8	`SELECT ssn, salary FROM employee WHERE fname = 'Ahmad'`	Select the ssn and salary of each employee whose first name is "Ahmad."	`SELECT ssn, dept_no FROM salesperson WHERE name = 'John Smith'`	Select the ssn and department number of each salesperson whose name is "John Smith."
9	`SELECT * FROM employee WHERE fname = 'Ahmad'`	Select all attributes of each employee whose first name is "Ahmad."	`SELECT * FROM salesperson WHERE name = 'John Smith'`	Select all attributes of each salesperson whose name is "John Smith."
10	`SELECT * FROM employee WHERE salary > 20000 AND fname = 'Ahmad'`	Select all attributes of each employee whose salary is greater than $20,000 and first name is "Ahmad."	`SELECT * FROM salesperson WHERE start_year > '2012' AND name = 'John Smith'`	Select all attributes of each salesperson whose start year is later than 2012 and name is "John Smith."
11	`SELECT DISTINCT fname FROM employee WHERE fname = 'Ahmad'`	Select the distinct first names of all employees whose first names are "Ahmad."	`SELECT DISTINCT start_year FROM salesperson WHERE name = 'John Smith'`	Select the distinct start years of all salespersons whose names are "John Smith."
12	`SELECT DISTINCT fname, salary FROM employee WHERE fname = 'Ahmad'`	Select the distinct values of the first names and salaries of all employees whose first names are "Ahmad."	`SELECT DISTINCT start_year, dept_no FROM salesperson WHERE name = 'John Smith'`	Select the distinct values of the start years and department numbers of all salespersons whose names are "John Smith."
13	`SELECT DISTINCT * FROM employee WHERE fname = 'Ahmad'`	Select the distinct values of all attributes of all employees whose first names are "Ahmad."	`SELECT DISTINCT * FROM salesperson WHERE name = 'John Smith'`	Select the distinct values of all attributes of all salespersons whose names are "John Smith."
14	`SELECT salary, dname FROM employee, department WHERE department.dnumber = employee.dno`	Select the salary of each employee and the name of each department where the employee works for the department.	`SELECT salesperson.name, trip.to_city FROM salesperson, trip WHERE salesperson.ssn = trip.ssn`	Select the name of each salesperson and the destination of each trip where the salesperson takes the trip.
15	`SELECT salary, dname FROM employee, department WHERE department.dnumber = employee.dno AND fname = 'Ahmad'`	Select the salary of each employee and the name of each department where the employee works for the department and the employee's first name is "Ahmad."	`SELECT salesperson.name, trip.to_city FROM salesperson, trip WHERE salesperson.ssn = trip.ssn AND trip.from_city = 'Atlanta'`	Select the name of each salesperson and the destination of each trip where the salesperson takes the trip and the trip's origin is "Atlanta."

type of the attributes differs; this information propagates to the constant value ("$20,000" vs "2012") and the operator ("greater" vs "later") expressions. These results show that our rules are capable of handling different schemas and data models while being defined independently of both.

7.3 Problem Guidance

Our last experiment is motivated by problem guidance. Instructors choose queries to teach specific concepts, in isolation and in combination. We assume a partial ordering of queries based on syntactic complexity. If a student is not able to answer a particular problem correctly, they can instead be given a simpler problem that incorporates fewer concepts simultaneously. For this experiment, we created 20 queries of decreasing complexity and paraphrased them with our tool.

7.4 Rule Categorization

Our system is currently comprised of 25 rules, of which 7 are analytic rules, 6 are grammar rules and 12 are lowering rules. There are 44 unique queries translated in the previous evaluation sections. Figure 14 shows the distribution of rule usage by query translations. The x-axis shows the number of distinct rules[7] used for a particular translation, while the y-axis shows the number of queries that used that number of rules. For example, the first bar from the left indicates that there were 10 translations that used 4 rules. No single query translation used more than 14 rules and there is only one translation that requires this many. The majority of query translations use 8 or fewer rules to translate.

Figure 15 shows the number of query translations that each rule applied to, with a mean of approximately 13. There are a small number of rules used in almost every translation, which are responsible for producing the overall structure (Select List Format) and lowering the most common token types (Select Label, Entity Ref Lowering, and Attribute Literal

[7] Multiple uses of the same rule are only counted once.

Table 3: Problem guidance experiment.

	Query	Generated NL
1	`SELECT * FROM employee AS E, employee AS S WHERE E.super_ssn = S.ssn AND E.salary > 20000 AND E.fname = 'Ahmad'`	Select all attributes of each employee and their supervisor where their salary is greater than $20,000 and their first name is "Ahmad."
2	`SELECT E.* FROM employee AS E, employee AS S WHERE E.super_ssn = S.ssn AND E.salary > 20000`	Select all attributes of each employee *e* where employee *e2* supervises *e* and *e*'s salary is greater than $20,000.
3	`SELECT * FROM employee AS E, employee AS S WHERE E.super_ssn = S.ssn AND E.salary > 20000`	Select all attributes of each employee and their supervisor where their salary is greater than $20,000.
4	`SELECT E.fname, S.fname FROM employee AS E, employee AS S WHERE E.super_ssn = S.ssn AND E.salary > 20000`	Select the first names of each employee and their supervisor where their salary is greater than $20,000.
5	`SELECT E.fname, S.fname FROM employee AS E, employee AS S WHERE E.super_ssn = S.ssn`	Select the first names of each employee and their supervisor.
6	`SELECT salary, dname FROM employee, department WHERE department.dnumber = employee.dno AND fname = 'Ahmad'`	Select the salary of each employee and the name of each department where the employee works for the department and the employee's first name is "Ahmad."
7	`SELECT salary, dname FROM employee, department WHERE department.dnumber = employee.dno`	Select the salary of each employee and the name of each department where the employee works for the department.
8	`SELECT DISTINCT * FROM employee WHERE fname = 'Ahmad'`	Select the distinct values of all attributes of all employees whose first names are "Ahmad."
9	`SELECT DISTINCT fname, salary FROM employee WHERE fname = 'Ahmad'`	Select the distinct values of the first names and salaries of all employees whose first names are "Ahmad."
10	`SELECT DISTINCT fname FROM employee WHERE fname = 'Ahmad'`	Select the distinct values of the first names of all employees whose first names are "Ahmad."
11	`SELECT * FROM employee WHERE salary > 20000 AND fname = 'Ahmad'`	Select all attributes of each employee whose salary is greater than $20,000 and first name is "Ahmad."
12	`SELECT * FROM employee WHERE fname = 'Ahmad'`	Select all attributes of each employee whose first name is "Ahmad."
13	`SELECT ssn, salary FROM employee WHERE fname = 'Ahmad'`	Select the ssn and salary of each employee whose first name is "Ahmad."
14	`SELECT ssn FROM employee WHERE fname = 'Ahmad'`	Select the ssn of each employee whose first name is "Ahmad."
15	`SELECT DISTINCT * FROM employee`	Select the distinct values of all attributes of all employees.
16	`SELECT DISTINCT fname, salary FROM employee`	Select the distinct values of the first names and salaries of all employees.
17	`SELECT DISTINCT fname FROM employee`	Select the distinct first names of all employees.
18	`SELECT * FROM employee`	Select all attributes of each employee.
19	`SELECT ssn, salary FROM employee`	Select the ssn and salary of each employee.
20	`SELECT ssn FROM employee`	Select the ssn of each employee.

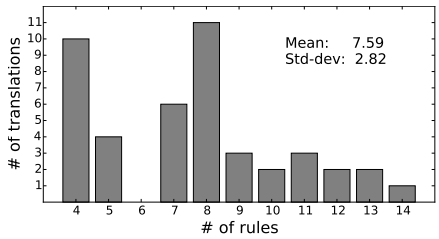

Figure 14: Rule distribution by query translation.

Label.) Application of the remaining rules is highly dependent on the structure of the query being translated. Grammar rules tend to have wider applicability, but infrequent application does not mean a rule is not important. For example, the Entity Ref Needs Id Rule is only applied in two translations, yet this rule prevents ambiguity when two or more entity instances would use the same label; if it were not applied, then these translations would be confusing. A particular strength of our approach is that this initial rule base can be easily extended to handle additional query constructs or produce more sophisticated verbalizations. Lastly, there are four rules that are not exercised by our evaluation queries; these rules handle cases that occur in our deployed system or would occur if more advanced rules were not applied first.

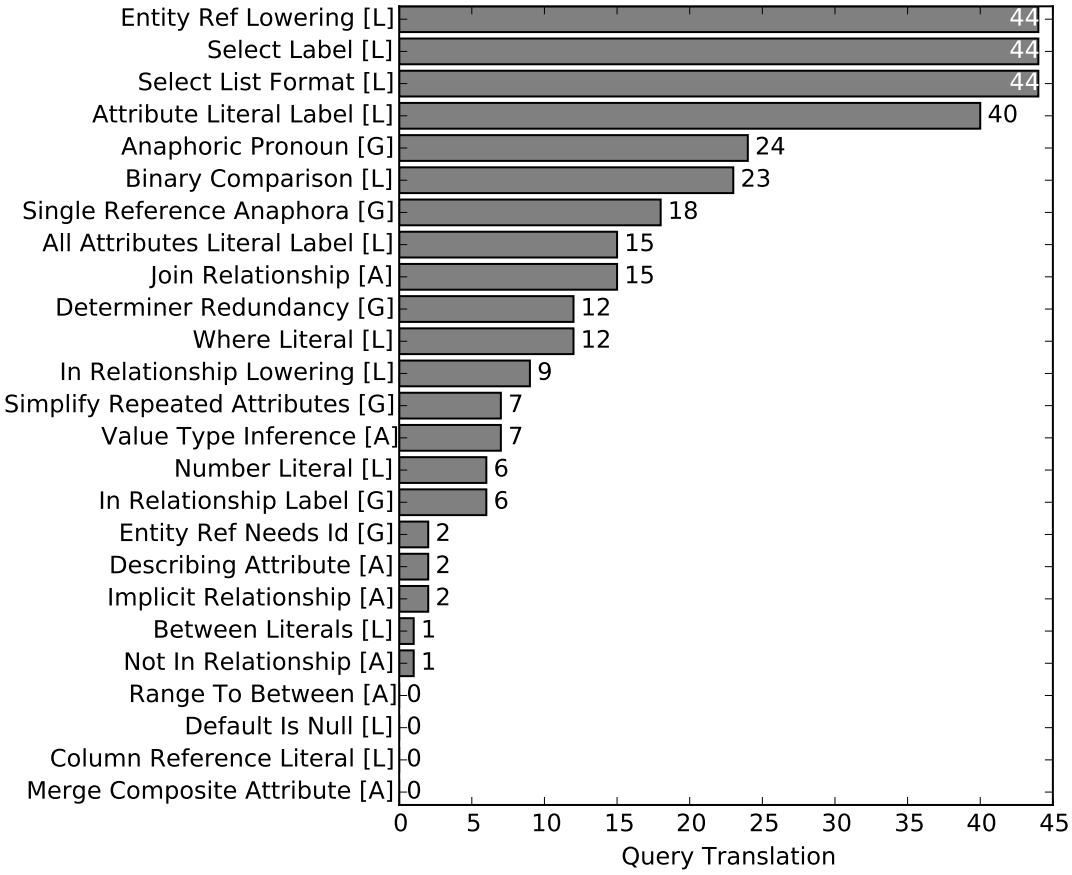

Figure 15: Total rule applications.

CHAPTER VIII: CONCLUSION, RELATED WORK, AND FUTURE WORK

8.1 Related Work

SPARQL2NL [24] translates queries written in SPARQL, a query language for Resource Description Framework (RDF) databases, to natural language. SPARQL2NL is also rule-based, but generates its output in three natural language dependency trees corresponding to parts of the SPARQL query, which include the original variable names; it then applies rules that attempt to simplify this structure. Labels and verbalizations are taken by querying an RDF database with heuristic fallback. In contrast, our approach treats the query and output structure uniformly, is completely static, and uses a conceptual model to derive domain knowledge and base phrases. Having a query's target database also contain encoding of domain knowledge is a characteristic of some RDF databases, such as those that encode ontologies, but does not generalize to other database systems.

ELFS [22] is an early system that also generates English text to describe SQL queries. ELFS preprocesses SQL queries into pseudo-queries which are passed to a text generation component. ELFS uses a fixed output strategy and requires a verbalization database for each schema it handles. Our approach is extensible by rule addition, does not require preprocessing, and uses a conceptual model for domain knowledge and base phrases.

There is recent work on automatically generating new problems, given an example problem or certain problem features, for a variety of subject domains. In contrast, the input to our system is a solution (which has a logical form) to the desired problem description. Furthermore, the prior work has focused on generating problems at a logical, symbolic, or numerical level. In contrast, we generate a succinct and natural English description corresponding to the input logical description (a database query). We briefly summarize some of this prior work below.

In [29], the authors describe a problem generation technique for generating problems that are similar in structure to a given algebraic identity proof problem. They leverage continuity of the underlying domain of algebraic expressions, and use extension of polynomial identity testing to check the correctness of a generated problem candidate on a random input. In [3], the authors give a problem generation technique for the procedural domain, which includes problems commonly found in middle-school math curriculums, such as subtraction and greatest common divisor computation. Their technique leverages test input generation techniques to generate problems that explore various paths in the procedure that the student is expected to learn. In [1], the authors address problem generation for natural deduction problems. The underlying technique involves offline generation of a Universal Proof Graph (a hypergraph that represents all possible inference rule applications over propositions of bounded size) and then traversal of this graph to generate problems with specific features.

A lot of work exists on generating automated feedback and grading for programming assignments; a survey can be found in [19]. In more recent work, [30] presents a SAT-based search technique to generate repair-based feedback for functional correctness, and [16] gives a trace-comparison-based technique to provide feedback on performance problems. In contrast, our technique facilitates a very different kind of feedback by describing to students what their solution is doing. This can be useful when students misread the original problem description or are confused about what their solution achieves.

Recently, [2] proposed a feedback generation technique for automata construction problems. Their systems generates three kinds of feedback: counterexamples, repair to a nearest correct solution, and a description of what the student's solution is doing. The latter approach is closer to the kind of feedback that we generate. However, while we focus on generating succinct and natural English descriptions of a given logical form (a database query), [2] focuses on generating a logical description for the student's automata that is close to the logical description of the original problem description.

There is extensive work on natural language interfaces to databases (NLIDB) [4]. NaLIX [21, 20] presents a natural language query interface to an XML database using the structure of the natural language parse tree derived from the user description. PRECISE [18, 25, 26] translates semantically-tractable NL questions into corresponding SQL queries by matching tokens identified in the user description with the schema of the database. In contrast, we study the reverse problem that requires techniques based on rewrite rules and logical foundations as opposed to statistical reasoning that is common in work on NLIDB.

8.2 Future Work

8.2.1 Open problems

- Query Equivalency: Queries can come in many alternative yet equivalent forms due to inherent associative and commutative properties. This presents a problem to this area of research as the ordering of the query construct can drastically affect the outcome of generated sentences. This is largely due to how these constructs are parsed and fed into the inevitable tree structure. SPARQL2NL begins to solve this problem in its preprocessing stage, wherein it converts the query to disjunctive normal form, but is still prone to problems with reordering and primarily scope manipulation. For instance, if the query was manipulated such that a particular variable had the possibility of two separate labels, for instance “People" and “Employee", and the use of one of these labels over another was dependent on scope, SPARQL2NL’s approach would choose arbitrarily according to their described method because its pre-processing does not uphold or define scoping.

- Natural Language Equivalency: It is a non-trivial matter to produce alternative, equivalent expressions of a query in formal language. However, it could be described as crucial depending on the setting. By producing deterministic results, systems risk

falling into a robotic subset of English. In addition, suppose a user does not understand a particular derived feedback and wants to ask for the sentence to be rephrased. These sort of non-deterministic generations are completely unexplored by all of the research efforts mentioned within section 1.

- Complex Queries with Simple Generations: It is often the case that, due to the nature of a particular dataset, a user might write a particularly complex query using sub-expressions and aggregations which can be expressed in natural language using relatively few words. Conquering this sort of complexity mapping problem wherein the complexity of the query's parse tree in turn determines the complexity of the natural language tree still plagues the effort. SPARQL2NL begins to solve this problem by presenting its post processing phase which first expands the sentence then shrinks it with the assumption that it was at its maximal complexity. However, it still runs into problems with sub-expressions that cause it to generate multiple sentences (e.g. "The employee ... Additionally, who works for ... Additionally, who ... ", etc.).

8.2.2 Future avenues

- Syntactic Pairing: Natural language to query generation has received a much higher level of research efforts than the opposing idea mentioned in this paper. However, these areas of research are actually quite similar and could benefit each other by sharing techniques. For instance, it is rather easy to see that syntactic pairing to perform the generation of SQL queries could be adapted to generate varying English sentences by performing manipulations to their syntax trees.

- Discovering and Correcting Nonsensical Generations: Each method presented in section 1 makes the same assumption in the end: their generation was correct. However,

it is often more useful to ask: could my algorithm have done better? Machine learning is a huge topic today, and part of the reason for that is its auto-tuning capability. Perhaps if these systems asked users if their generation was nonsensical, it could guide the algorithm from making particular choices under certain conditions (e.g. use “employee" rather than “people" given a particular query structure.).

- Learning the Dataset: Yet another problem that each of these methods have is the reliance on a huge amount of foreknowledge. While this can be considered practical in the realm of databases, where database administrators undergo huge efforts to accurately represent datasets, it may not hold for smaller subsets that are also prone to being queried. File systems, for example, usually have large amounts of data that isn't nearly as structured, tagged, or appended as what these systems presume. However, these sorts of systems are vastly more common and querying across them is similarly complex. To expand into these areas, researchers in this area might look to efforts to 'learn' datasets (e.g. start from no prior knowledge to part-of-speech tagging, semantic labeling, entity recognition, etc.).

8.3 Conclusion

We presented a rule-based graph-rewriting algorithm for paraphrasing SQL queries in English. We provided a set of concrete rules that we implemented in a real system and showed their effectiveness at translating queries for applications in intelligent tutoring systems. Our results, while encouraging, leave a number of avenues for future work.

First, we devised a relatively small number of rules with a focus on the educational domain. We can extend our rule base to handle additional constructs, such as nested queries, or increase the sophistication of complex English expressions. Second, our current system supports SQL queries and uses ER diagrams as conceptual models; we will investigate generalizing our framework to other logical forms and conceptual models, e.g. Excel spreadsheet formulas. Third, our implementation uses manually-configured precedence to control

the order of rules; instead, machine learning techniques could be used to assign precedence to rules. Finally, we created our rule set based on insights we gained from observing example queries. We believe that new rules could be synthesized automatically based on a corpus of example queries and human-created translations.

REFERENCES

[1] U. Z. Ahmed, S. Gulwani, and A. Karkare. Automatically generating problems and solutions for natural deduction. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1968–1975, 2013.

[2] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of DFA constructions. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1976–1982, 2013.

[3] E. Andersen, S. Gulwani, and Z. Popovic. A trace-based framework for analyzing and synthesizing educational progressions. In *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 773–782, 2013.

[4] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural language interfaces to databases - An introduction. *CoRR*, pages 29–81, 1995.

[5] K. R. Apt, H. A. Blair, and A. Walker. *Towards a Theory of Declarative Knowledge*. IBM TJ Watson Research Center, 1986.

[6] B. Bishop and F. Fischer. IRIS - integrated rule inference system. In *International Workshop on Advancing Reasoning on the Web: Scalability and Commonsense (ARea 2008)*, 2008.

[7] S. Brass and C. Goldberg. Semantic errors in SQL queries: A quite complete list. *Journal of Systems and Software*, 79(5):630 – 644, 2006.

[8] H. C. Chan, K. K. Wei, and K. L. Siau. The effect of a database feedback system on user performance. *Behaviour & Information Technology*, 14(3):152–162, 1995.

[9] P. P.-S. Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[10] M. P. Consens and A. O. Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 404–416, New York, NY, USA, 1990. ACM.

[11] A. Copestake and K. S. Jones. Natural language interfaces to databases. *The Knowledge Engineering Review*, 5(04):225–249, 1990.

[12] R. Elsmari and S. Navathe. Fundamentals of database systems, 2000.

[13] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra. Data-guided repair of selection statements. In *ICSE*, pages 243–253, 2014.

[14] S. Gulwani. Example-based learning in computer-aided STEM education. *Commun. ACM*, 57(8):70–80, 2014.

[15] S. Gulwani and M. Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. SIGMOD, 2014.

[16] S. Gulwani, I. Radicek, and F. Zuleger. Feedback generation for performance problems in introductory programming assignments. In *FSE*, 2014.

[17] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 405–418, New York, NY, USA, 2008. ACM.

[18] G. G. Hendrix, E. D. Sacerdoti, D. Sagalowicz, and J. Slocum. Developing a natural language interface to complex data. *TODS*, 1978.

[19] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *10th Koli Calling International Conference on Computing Education Research*, pages 86–93, 2010.

[20] Y. Li, H. Yang, and H.Jagadish. NaLIX: An interactive natural language interface for querying XML. In *SIGMOD*, 2005.

[21] Y. Li, H. Yang, and H.Jagadish. Constructing a generic natural language interface for an XML database. In *EDBT*, 2006.

[22] W. S. Luk and S. Kloster. ELFS: English language from SQL. *ACM Trans. Database Syst.*, 11(4):447–472, Dec. 1986.

[23] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19(2):313–330, 1993.

[24] A.-C. Ngonga Ngomo, L. Bühmann, C. Unger, J. Lehmann, and D. Gerber. Sorry, I don't speak SPARQL: Translating SPARQL queries into natural language. In *Proceedings of the 22nd international conference on World Wide Web*, pages 977–988. International World Wide Web Conferences Steering Committee, 2013.

[25] A.-M. Popescu, A. Armanasu, O. Etzioni, D. Ko, and A. Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *COLING*, 2004.

[26] A.-M. Popescu, O. Etzioni, and H. Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, 2003.

[27] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006.

[28] M. A. Rodriguez and P. Neubauer. The graph traversal pattern. In *Graph Data Management: Techniques and Applications.*, pages 29–46. 2011.

[29] R. Singh, S. Gulwani, and S. K. Rajamani. Automatically generating algebra problems. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[30] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013.

[31] J. P. Titlow. 15 programming skills most coveted by employers. Apr. 2013.

[32] C. Welty. Correcting user errors in SQL. *International Journal of Man-Machine Studies*, 22(4):463 – 477, 1985.